Entwurf und Implementierung eines Baukastens für Verteilte Algorithmen

Tim Gottwald

Diplomarbeit

November 2003

Betreuung:

Prof. Dr. Rainer Oechsle

Fachbereich Design und Informatik
Fachhochschule Trier
University of Applied Sciences

FACHHOCHSCHULE TRIER
UNIVERSITY OF APPLIED SCIENCES
Fachbereich
DESIGN UND INFORMATIK

Autor:	**Tim Gottwald**
Titel:	**Entwurf und Implementierung eines Baukastens für Verteilte Algorithmen**
Studiengang:	**Angewandte Informatik**
Betreuung:	**Prof. Dr. Rainer Oechsle**

November 2003

Es wird hiermit der Fachhochschule Trier (University of Applied Sciences) die Erlaubnis erteilt, die Arbeit zu nicht-kommerziellen Zwecken zu verteilen und zu kopieren.

Unterschrift des Autors

© Copyright: Tim Gottwald (2003)

Ein besonderer Dank gilt:

Prof. Dr. Rainer Oechsle für viele wertvolle Hinweise und eine gute Betreuung.

Meiner Freundin Angela Otting, die so tapfer war und diese Arbeit Korrektur gelesen hat, und in schwierigen Phasen für die nötige Aufheiterung sorgte.

Daniel Röder, der das Programm einem ersten praktischen Test unterzogen und die Tauglichkeit des Benutzerhandbuchs überprüft hat.

Meinem Kommilitonen Ralph Weires, mit dessen Hilfe viele Probleme gelöst werden konnten, und der mir vor allem im Bereich Software-Testen viele nützliche Tipps geben konnte.

Den „Freunden der Robotik", Florian Adolf, Bernhard Kloss und Dirk Simon für Zeiten der Ablenkung und des Teetrinkens, in denen ich zusätzlich noch einiges über Bildverarbeitung lernen durfte.

Meinen Eltern, die mich in den letzten 4 Jahren so toll (nicht nur finanziell:) unterstützt haben.

Kurzfassung

Verteilte Algorithmen beschäftigen sich mit der Lösung von Problemen, die in verteilten Systemen auftreten können. Unter diese Probleme fallen z.B. die Uhrensynchronisation in Netzwerken, globale Zustände in verteilten Systemen oder Koordinations- und Übereinstimmungsprobleme.

Ziel dieser Arbeit ist es, ein Baukastensystem zu entwickeln und zu realisieren, das viele Mittel bereitstellt, um verteilte Algorithmen zu implementieren. Der Anwender dieses Baukastens soll in der Lage sein, die angebotenen Elemente schnell und intuitiv einzusetzen. Ein wesentlicher Aspekt dabei ist eine hohe Erweiterbarkeit, die die Festlegung der Grenzen dieses Systems erschwert. Neben der einfachen Entwicklung verteilter Systeme sollen diese zusätzlich grafisch visualisiert werden. Die grafische Darstellung behebt oft Verständnisprobleme des Ablaufs eines verteilten Algorithmus.

Ergebnis der Arbeit sind eine Klassenbibliothek und ein Programm, die alle gesteckten Ziele beinhalten. Die Bibliothek verfügt über Klassen, aus denen sich ein Entwickler die von ihm benötigten herausgreifen und erweitern kann, um mit den darin enthaltenen Methoden die unterschiedlichsten Algorithmen zu implementieren. Diese Klassen können dann in das Programm DisASTer - *Distributed Algorithms Simulation Terrain* - geladen werden. Hauptaufgabe dieses Programms ist die Ablaufsteuerung der Algorithmen sowie deren Visualisierung. Die nachrichtengesteuerte Simulation kann dabei automatisch oder manuell gesteuert werden. Die Visualisierung erfolgt z.B. in Form eines animierten Sequenzdiagramms.

Inhaltsverzeichnis

1 Einleitung **1**

2 Aufgabenstellung **3**

3 Verteilte Algorithmen **6**
3.1 Physikalische und logische Uhren 6
3.2 Globale Zustände 13
3.3 Koordination und Übereinstimmung 15

4 Überblick und Grobkonzept **34**
4.1 Programmkomponenten 34
4.2 Eigenschaften verteilter Algorithmen 34
4.3 Kontrolle und Steuerung 36
4.4 Visualisierung 39

5 Benutzung des Programmes **40**
5.1 Inhalt und Installation 40
5.2 Die Bibliothek 42
5.3 Die Oberfläche 52

6 Entwurf **71**
6.1 Eigenschaften verteilter Algorithmen 71
6.2 Erkenntnisse für den eigenen Entwurf 72
6.3 Model 75
6.4 View 83
6.5 Controller 85
6.6 Grafische Benutzeroberfläche (GUI) 87

7 Implementierung und Test **91**
7.1 Topologieeditor 91
7.2 Visualisierung und Animation 93
7.3 Klassenladen 95
7.4 Reflection und Pseudopolymorphismus 96
7.5 Oberflächengestaltung 98
7.6 Speichern von Umgebungszuständen 100
7.7 Verteilung mit Ant 101
7.8 Hilfesystem 102
7.9 Test 107

8 Beispiele **111**
8.1 Logische Uhren 111
8.2 Koordination und Übereinstimmung 116

9 Verwandte Projekte **135**
9.1 Vorstellung der Projekte 135
9.2 Zusammenfassung und Vergleich 137

10 Zusammenfassung und Ausblick **140**

Literaturverzeichnis **142**

A Klassenübersicht **144**

1 Einleitung

„Verteilte Algorithmen“ ist ein Teilgebiet aus der Vorlesung „verteilte Systeme“. Die Schwierigkeit dieses Themas liegt in der umständlichen Umsetzung von Übungen. Es muss eine Umgebung geschaffen werden, in der die Algorithmen ablaufen können. Dabei beschränkt sich die Ausgabe dieser Umgebung häufig auf Textausgaben auf der Konsole, die das Verständnis für die Arbeitsweise der einzelnen Algorithmen nicht fördern.

Diese Arbeit beschäftigt sich mit der Entwicklung und Umsetzung eines Systems, das diese Übungen beträchtlich vereinfachen soll. Dabei handelt es sich um einen Baukasten und ein Programm, die es dem Anwender erlauben, verteilte Algorithmen in Java zu beschreiben. Aus diesem Baukasten können Klassen erweitert werden, die Methoden bereitstellen, solche Algorithmen zu implementieren. Dabei muss sich der Anwender nur auf die Umsetzung der Algorithmen konzentrieren. Die Simulation und Visualisierung wird ihm von dem beiliegenden Programm abgenommen.

Im 2. Kapitel wird die genaue Aufgabenstellung dieser Arbeit vorgestellt, in der die zu erreichenden Ziele gesteckt werden. Zusätzlich findet sich in diesem Kapitel eine Anforderungsanalyse, in der die Eigenschaften und Punkte aufgeführt werden, die im Laufe der Arbeit erfüllt werden sollen.

Das darauf folgende Kapitel, führt den Leser in das Thema ein. Dort werden Algorithmen zu den Themen Uhrensynchronisation, globale Zustände, Koordinationsprobleme, wie Festlegung einer Nachrichtenreihenfolge, Wahl eines Anführers oder Übereinstimmungsprobleme, z.B. das der byzantinischen Generäle, vorgestellt.

Kapitel 4 gibt einen Überblick über das entstandene Programm und führt den Leser in die entscheidenden Konzepte ein.

Das Benutzerhandbuch des entstandenen Programms findet sich in Kapitel 5. Dort werden die Funktionen des Programms DisASTer sowie die Benutzung der entwickelten Bibliothek beschrieben.

Kapitel 6 widmet sich dann dem objektorientierten Entwurf der Arbeit. Dieser besteht im Wesentlichen aus dem Model-View-Controller-Paradigma und stellt die jeweiligen Komponenten vor.

In Kapitel 7 werden besondere Programmteile herausgepickt, deren Implementierung entweder problematisch oder trickreich ist. Das Kapitel wird mit der Beschreibung der Tests dieses Programms abgeschlossen.

Um einen weiteren Einblick in den Funktionsumfang von DisASTer zu geben, werden in Kapitel 8 einige Beispiele aufgeführt, die zeigen, wie mit der entstandenen Bibliothek gearbeitet werden kann.

DisASTer ist nicht das einzige Projekt, dass sich mit der Simulation verteilter

Algorithmen beschäftigt hat. Einige vergleichbare Projekte, die zu Beginn dieser Arbeit untersucht wurden, finden sich in Kapitel 9.

In Kapitel 10 werden noch einmal die Ergebnisse dieser Diplomarbeit reflektiert und ein Ausblick auf die weitere Entwicklung und den Einsatz von DisASTer gegeben.

2 Aufgabenstellung

Das Thema „verteilte Algorithmen“ bildet einen Teil der Vorlesung „verteilte Systeme“. Kurz und knapp beschrieben, lösen verteilte Algorithmen Probleme innerhalb eines verteilten Systems. Was genau unter diesem Begriff zu verstehen ist, wird im nächsten Kapitel genauer erklärt.

Verteilte Algorithmen sind bei dem immer häufigeren Einsatz von Netzwerken und dementsprechend auch dem Internet allgegenwärtig. Allerdings agieren sie häufig im Hintergrund und ohne Kenntnis der Benutzer. Dabei handelt es sich um Mechanismen, die zunächst sehr schwer zu verstehen sind. Praktische Übungen sind bislang nur sehr umständlich zu lösen, da der Aufwand, sie zu realisieren sehr hoch ist. Dadurch fördern diese Übungen nicht gerade dem Verständnis sondern schrecken eher ab. „Ein Bild sagt mehr als tausend Worte“. Dieses Sprichwort trifft auch nur bedingt auf das Thema „verteilte Algorithmen“ zu. Um diese darzustellen werden häufig Sequenzdiagramme verwendet. Bei Algorithmen, bei denen ein hoher Nachrichtenaustausch nötig ist, nehmen diese Diagramme aber beträchtliche Ausmaße an und führen häufig dazu den Betrachter zunächst einmal zu erschlagen.

Diese Arbeit soll helfen, den Benutzer für das Thema „verteilte Algorithmen“ zu sensibilisieren, die Darstellung zu verbessern und vor allem praktische Übungen zu erleichtern. Um diesem Ziel gerecht zu werden, soll eine Java-Bibliothek entworfen werden, mit deren Hilfe das Erzeugen verteilter Algorithmen zu Übungszwecken kein Problem mehr darstellt. Daneben soll ein Programm entwickelt werden, dass diese Algorithmen ausführen und in einer übersichtlichen Form visualisieren kann. Die Arbeitsweise soll der eines Baukasten entsprechen, wobei der Nutzer der Bibliothek und des Programms eine Auswahl aus Klassen und Methoden erhält, mit denen er einen verteilten Algorithmus implementieren kann.

Die Bibliothek sollte folgenden Anforderungen gerecht werden:

- Die angebotenen Klassen und deren Methoden sollten schnell verstanden und leicht eingesetzt werden können. Dadurch soll dem Anwender eine lange Einarbeitungszeit erspart bleiben, die ihm dann zur Entwicklung der Algorithmen zur Verfügung steht.

- Um das Programm auch in Gebieten einzusetzen, an die zur Zeit noch nicht gedacht wird, sollte es einen hohen Grad an Erweiterbarkeit besitzen.

- Es sollten möglichst viele Algorithmen verwirklicht werden können. Das setzt entweder eine sehr umfangreiche oder eine sehr allgemein gehaltene Bibliothek voraus. Eine allgemein gehaltene Bibliothek gibt dem Benutzer eher die Möglichkeit, Erweiterungen vorzunehmen und ist auch schneller

verstanden. Einziger Nachteil ist, dass der Programmieraufwand und die hierfür nötige Eigeninitiative ein wenig ansteigt.

Sind die Anforderungen an die zu entwerfende Bibliothek doch eher abstrakt, werden die Anforderungen an das Programm schon ein wenig konkreter:

- Das Programm sollte eine gute Ergonomie besitzen, Anordnung und Funktionsweise der Oberflächenelemente sollten dem aus anderen Programmen üblichen Aufbau entsprechen. Der Anwender sollte Elemente oder Strukturen vorfinden, die ihm allgemein bekannt sind, um eine schnelle und intuitive Bedienung zu garantieren.

- Das Erstellen der verteilten Systeme, in denen die Algorithmen eingesetzt werden, sollte ebenfalls möglichst einfach und intuitiv erfolgen. Am besten wäre eine grafische Darstellung des Systems, beispielsweise in Form der Topologie eines Netzwerks.

- Um auf die erstellten Systeme jederzeit zugreifen zu können und um sie nicht bei jedem Programmaufruf neu konfigurieren zu müssen, sollten sie gespeichert werden können. Die gespeicherten Systeme bzw. Algorithmen können dadurch auch ausgetauscht und auf anderen Rechnern ausgeführt werden.

- Die vom Benutzer erzeugten Klassen und Erweiterungen sollten einfach in das Programm eingebunden werden können.

- Die Ablaufsteuerung der Simulation sollte dem Anwender viele Möglichkeiten bieten. Zum einen soll er die Möglichkeit haben, Algorithmen automatisch ablaufen zu lassen, zum anderen soll er in den Ablauf eingreifen und ihn verändern können. Dies soll mit einer manuellen Steuerung ermöglicht werden. Um auch hier eine möglichst schnelle und intuitive Bedienung zu garantieren, könnten bekannte Funktionen wie Start, Stopp, Pause und Einzelschritt verwendet werden.

- Die wichtigste Aufgabe des Programms ist allerdings die Darstellung und Visualisierung der Algorithmen. Diese sollte in einer bekannten Form stattfinden. Wie oben bereits erklärt, geschieht dies oft über Sequenzdiagramme. Eine Alternative dazu wäre eine Ansicht auf die Topologie des verteilten Systems. Da die starren Ablaufdiagramme auf den ersten Blick zunächst sehr unübersichtlich wirken, könnte die Visualisierung in einer bewegten Animation stattfinden. Diese Art der Darstellung erhöht die Übersichtlichkeit und erleichtert das Verständnis der Arbeitsweise eines verteilten Algorithmus. Um die Übersichtlichkeit um noch einen weiteren Faktor zu

steigern, sollte diese Visualisierung vom Anwender angepasst werden können.

- Um das Programm in möglichst vielen Themengebieten einsetzen zu können, sollte es auch in hohem Grad erweiterbar sein. Hierzu gehört die Erweiterbarkeit des Funktionsumfangs und die der Darstellungen.

Neben den gerade vorgestellten Anforderungen, denen das Programm und die Bibliothek auf jeden Fall gerecht werden sollten, gibt es noch einige Punkte, die sich lohnen würden, integriert zu werden. Diese Nice-To-Haves müssen allerdings nicht um jeden Preis realisiert werden. Zu ihnen gehören:

- Das Einbinden von Jar-Archiven, die die Erweiterungen der Bibliothek beinhalten. Es könnte mit Hilfe eines Dateiauswahlfensters eine Klasse ausgewählt werden, die sich in einem Archiv befindet. Hierzu müsste der Klassenpfad angepasst werden und das Dateiauswahlwerkzeug müsste den Inhalt dieser Archive anzeigen können.

- Das Programm könnte netzwerkfähig gemacht werden, um die entwickelten Algorithmen in einem realen Netzwerk einsetzen zu können.

- Die Visualisierung könnte als Grafik oder sogar als Film exportiert werden, wodurch die erzielten Ergebnisse besser dokumentiert werden könnten. Außerdem könnten diese Filme oder Grafiken in Vorträgen, Vorlesungen oder Dokumentationen verwendet werden.

3 Verteilte Algorithmen

Dieses Kapitel gibt einen Einblick in die Thematik „verteilte Algorithmen“, um den Hintergrund und die Motivation dieser Arbeit zu vermitteln.

Verteilte Algorithmen finden sich in vielen verteilten System wieder, einige davon sogar im größten verteilten System, dem Internet. Damit sind verteilte Algorithmen allgegenwärtig, obwohl sie häufig unerkannt im Hintergrund fungieren. Viele von ihnen sind dazu da, Fehler zu vermeiden oder Probleme zu lösen, die in verteilten Systemen auftreten können. Dabei spielt die Zeit eine große Rolle. In diesem Zusammenhang stehen Probleme, wie Uhrensynchronisation, logische Zeit oder globale Zustände. Die Uhrensynchronisation der einzelnen Teinehmer eines Netzwerks ist oft die Voraussetzung, um weitere Probleme z.B. im Bereich Koordination und Übereinstimmung zu lösen. Unter diesen Punkt fallen zum Beispiel die Wahl eines Anführers oder der gegenseitige Ausschluss.

Auf die einzelnen Probleme und deren Lösungen wird in den folgenden Abschnitten eingegangen.

3.1 Physikalische und logische Uhren

Dieser Abschnitt beschäftigt sich mit dem Problem der Zeit. Dabei wird zunächst ein Unterschied zwischen der physikalischen und der logischen Zeit gemacht. Die physikalische Zeit bezieht sich auf die reale Zeit, die auf Uhren dargestellt wird. Mit Hilfe logischer Uhren soll, unabhängig von der jeweiligen physikalischen Zeit, festgelegt werden können, ob ein Ereignis vor einem anderen eintritt.

3.1.1 Uhrensynchronisation

Ein Problem in verteilten Systemen ist die Uhrensynchronisation. Jeder Prozess eines verteilten Systems besitzt eine physikalische Zeit. Diese muss aber nicht unbedingt identisch mit denen der anderen Prozesse sein, sondern es treten in der Regel Abweichungen auf. Um diese Synchronisationfehler zu beheben, gibt es zwei unterschiedliche Methoden:

1. die externe Synchronisation und
2. die interne Synchronisation.

Bei der externen Synchronisation wird eine externe Quelle, z.B. eine UTC-Quelle[1], verwendet; die interne Synchronisation hebt Abweichungen zwischen zwei Prozessen auf. Die interne Synchronisation ziehlt somit nicht auf die Genauigkeit der Uhren im Vergleich zu einer externen Quelle, sondern auf eine möglichst genaue Stimmigkeit zwischen den einzelnen Prozessen.

Um eine interne Synchronisation zu gewährleisten, wurden bereits mehrere Verfahren entwickelt.

Christian Bei der Methode von Christian wird ein Zeit-Server verwendet. Dieser erhält seine Zeit zunächst von einer externen Quelle. Coulouris, Dollimore und Kindberg beschreiben die weitere Arbeitsweise dieser Methode wie folgt:

> „Ein Prozess p fordert die Zeit in einer Nachricht m_r an und empfängt den Zeitwert t in einer Nachricht m_t (t wird in m_t zum letzten möglichen Zeitpunkt vor der Übertragung vom Computer von S eingetragen). Der Prozess p zeichnet die Gesamt-Round-Trip-Zeit T_{round} auf, die benötigt wurde, um die Anforderung m_r zu senden und die Antwort m_t zu empfangen. [...] Eine einfache Schätzung der Zeit, auf die p seine Uhr setzen soll, ist
>
> $$t + T_{round}/2,$$
>
> wobei vorausgesetzt wird, dass die vergangene Zeit gleichmäßig aufgeteilt wurde, bevor S t in m_t plaziert hat.“ ([Coulouris et. al. 02], S. 456])

Ein Nachteil dieser Methode ist die Abhängigkeit von einer zentralen Instanz, da diese unter Umständen ausfallen kann. Eine Abhilfe würde der Einsatz mehrerer Zeitserver schaffen, die gemeinsam angeschrieben werden und deren schnellste Antwort verwendet wird. Fehlerhafte Angaben eines Zeit-Servers oder sogar Fehler der externen Quelle werden in diesem Verfahren nicht berücksichtigt (vgl. [Coulouris et. al. 02], S. 456f).

BTP - Berkeley Time Protocol Gusella und Zatti haben einen Algorithmus entwickelt, um eine interne Synchronisation ohne einen zentralen Zeitserver zu bekommen. Dieser wurde unter Berkeley-UNIX-Systemen eingesetzt, wodurch er seinen Namen erhalten hat. Bei dieser Methode wird ein Prozess als Koordinations-Computer (*Master*) gewählt. Die anderen Prozesse werden als *Slave* bezeichnet. Der *Master* führt in gewissen Zeitabständen eine Umfrage bei

[1] UTC: Universal Time Coordinatet. Bezeichnung für eine für die gesamte Erde einheitliche Zeitskala.

den *Slaves* durch und erkundigt sich nach deren aktueller Zeit. Nachdem diese bei ihm angekommen sind, werden die aktuellen Zeiten der anderen Prozesse mit Hilfe der Round-Trip-Zeit, ähnlich der Methode Christians, geschätzt und inklusive seiner eigenen aktuellen Zeit gemittelt. Dieser Mittelwert überdeckt bereits die Tendenzen der einzelnen Uhren, zu schnell oder zu langsam zu laufen. Um mögliche Ausreißer auszuschließen, wird eine maximal erlaubte Round-Trip-Zeit festgelegt, die gleichzeitig die Genauigkeit des Algorithmus bestimmt. Ausreißer werden aus diesem Verfahren eliminiert, wodurch ein fehlertoleranter Mittelwert entsteht. Anstatt den einzelnen Prozessen die neue aktuelle Zeit zu senden, erhalten diese die Zeitdifferenz, die ausgeglichen werden muss. Dabei kann es sich sowohl um einen negativen als auch um einen positiven Wert handeln.

Die einzige Gefahr, die bei diesem Verfahren besteht, ist ein Ausfall des *Master*-Prozesses. Sollte dies der Fall sein, müssen die übriggebliebenen Prozesse einen neuen *Master* bestimmen. Algorithmen, die sich dieser Problematik widmen, werden in Abschnitt 3.3.2 beschrieben (vgl. [Coulouris et. al. 02], S. 457f).

NTP Network Time Protocoll Die beiden oben vorgestellten Algorithmen, BTP und die Methode von Christian, sind ausschließlich für den Einsatz in lokalen Netzwerken vorgesehen. Um Computer auch über das Internet zu synchronisieren, wurde das NTP-Protokoll entwickelt. Dabei verfolgt NTP folgende Ziele:

1. Synchronisierung mit einer UTC-Quelle über das Internet
2. Überstehen längerer Verbindungsunterbrechungen
3. Gewährleistung ausreichend häufiger Synchronisierung
4. Schutz vor Störung der Zeitdienste

Das NTP-Protokoll wird durch eine mehrschichtige Server-Architektur realisiert. Diese Schichten entstehen durch eine logische Hierarchie, beginnend mit einem Primär-Server als Wurzelelement und Sekundär-Servern in den darunter liegenden Schichten, wobei die Primär-Server direkt mit einer UTC-Quelle verbunden sind. Der Aufbau einer solchen Hierarchie wird durch Abbildung 3.1 dargestellt.

Eine Schwierigkeit, der NTP entgegenwirkt, ist, dass Server mit einer hohen Schichtennummer sehr wahrscheinlich ungenauer sind als Server aus einer übergeordneten Hierarchieebene. NTP löst das Problem über eine Bewertung der Qualität der Zeitnahmedaten.

Die Synchronisierung zwischen den einzelnen Servern und den anfragenden Computern kann über drei Modi erfolgen:

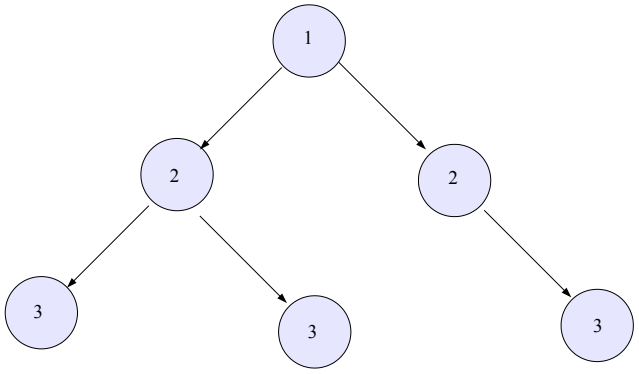


Abbildung 3.1: NTP-Hierarchie

1. Multicast-Modus : wird zur Serversynchronisierung in einem Hochgeschwindigkeits-LAN (Low Area Network) eingesetzt. Coulouris, Dollimore und Kindberg beschreiben diesen Modus wie folgt:

   > „Ein oder mehrere Server senden in bestimmten Zeitabständen Multicasts mit der Zeit an die Server, die auf anderen an das LAN angeschlossenen Computern ausgeführt werden, die ihre Uhren unter der Berücksichtigung einer kleinen Verzögerung setzen. Dieser Modus kann nur relativ geringe Genauigkeit erzielen, die aber für viele Aufgabenstellungen als ausreichend erachtet werden.“ ([Coulouris et. al. 02], S. 459)

2. Prozeduraufruf-Modus : wird eingesetzt, falls eine höhere Genauigkeit als bei der Multicast-Methode benötigt wird oder Multicast nicht unterstützt wird. Dieser Modus wird zur Synchronisierung innerhalb eines LANs oder zwischen benachbarten LANs verwendet. Die Arbeitsweise dieses Modus ähnelt erneut der Methode Christians. Nach einer Anforderung sendet ein Server seinen eigenen Zeitstempel.

3. Symetrische Modus : sorgt für die Synchronisierung der Server untereinander. Hierbei synchronisieren sich Server einer Hierarchieebene untereinander, um vor allem in den niedrigeren Schichten eine hohe Genauigkeit zu garantieren und diese auf längere Sicht aufrecht zu erhalten.

Der Austausch der Nachrichten ist in allen drei Modi unzuverlässig, da dort jeweils UDP eingesetzt wird (vgl. [Coulouris et. al. 02], S. 458ff).

3.1.2 Logische Uhren

Wie im vorangegangenen Abschnitt gezeigt, ist es sehr schwer, Computer in einem verteilten System zu synchronisieren. Das stellt vor allem dann ein Problem dar, wenn festgelegt werden muss, in welcher zeitlichen Beziehung Ereignisse zu einander stehen, die auf unterschiedlichen Prozessen auftreten. Mit dieser Problematik setzen sich logische Uhren auseinander. In Abbildung 3.2 ist eine Ereignisabfolge in einem verteilten System zu sehen.

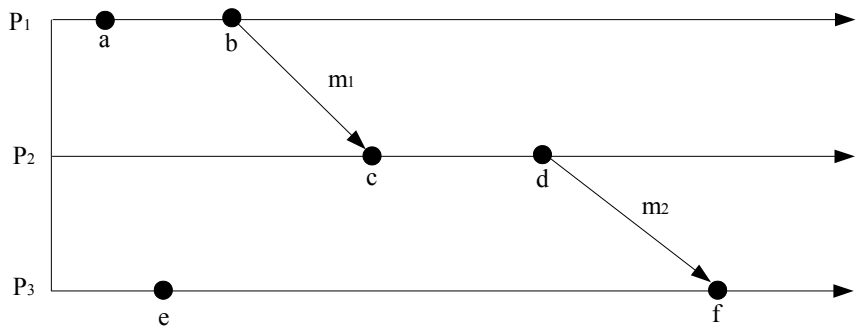


Abbildung 3.2: Ereignisabfolge in einem verteilten System

Aus der Abbildung wird deutlich, dass es Ereignisse gibt, die wiederrum andere Ereignisse auslösen können. Dies ist bei b und c oder d und f der Fall. Beispielsweise stellt Ereignis b bei P_1 das Senden der Nachricht m_1 dar. Das Eintreffen von m_1 wird bei P_2 als Ereignis c festgehalten. Damit stehen diese beiden Ereignisse in einer logischen und kausalen Reihenfolge. Im Gegensatz dazu sind die Ereignisse a und e unabhängig voneinander, was als nebenläufig bezeichnet wird. Um diesen beiden Ereignissen eine Reihenfolge zuzuordnen, werden logische Uhren verwendet.

Lamport Methode Lamport hat einen einfachen Mechanismus erfunden, mit dem es möglich ist, eine logische Zeit zu erstellen. Hierfür wird der sogennante Lamport-Zeitstempel verwendet. Dabei handelt es sich um einen monoton steigenden Softwarezähler, dessen Wert keinerlei Beziehung zu einer physikalischen Uhr haben muss. Jeder Prozess p_i eines verteilten Systems führt seine eigene logische Uhr, L_i. Das Ereignis e des Prozesses p_i wird dem Zeitstempel $L_i(e)$ zugewiesen. Um eine solche Lamport-Uhr zu verwenden, müssen folgende zwei Regeln befolgt werden:

LC1 L_i wird inkrementiert, bevor ein Ereignis in Prozess p_i veranlasst wird:

$$L_i := L_i + 1$$

LC2 a. Wenn ein Prozess p_i eine Nachricht m sendet, gibt er m auch den Wert $t := L_i$ mit.

b. Beim Empfang von $< m, t >$ berechnet ein Prozess p_j den Wert $L_j := max(L_j, t)$ und wendet dann LC1 an, bevor er dem Ereignis $receive(m)$ einen Zeitstempel zuweist.

Abbildung 3.3 zeigt das Beispiel aus Abbildung 3.2 mit logischen Uhren, wobei die Uhr jedes Prozesses p_i mit 0 initialisiert wurde.

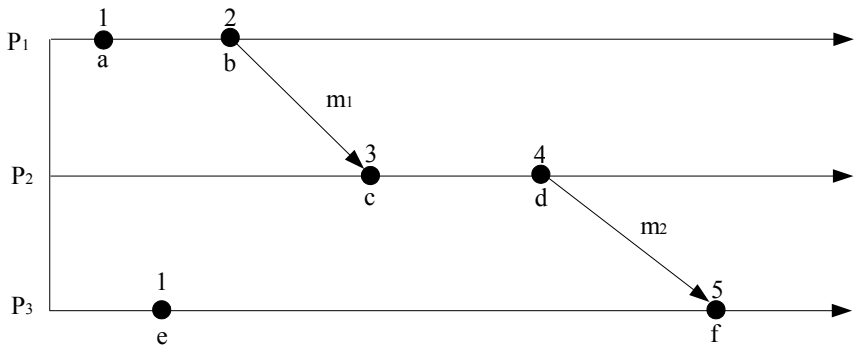


Abbildung 3.3: Ereignisabfolge mit logischen Uhren

Mit dieser Methode kann genau bestimmt werden, ob ein Ereignis vor einem anderen ausgelöst wurde. Weiter lässt sich festhalten, dass wenn ein Ereignis e das Ereignis e' ausgelöst hat ($e \rightarrow e'$), dann gilt:

$$e \rightarrow e' \Rightarrow L(e) < L(e')$$

Diese Feststellung lässt sich allerdings nicht umdrehen. Aus $L(e) < L(e')$ folgt nicht unmittelbar $e \rightarrow e'$.

Wie aus der Abbildung ebenfalls erkennbar, kann es passieren, dass zwei von einander unabhängige Ereignisse einen identischen Zeitstempel besitzen. Dies ist bei a und e der Fall. Um auch diese Ereignisse in eine logische Reihenfolge zu bringen, wird das Verfahren der Lamport-Uhr erweitert, indem ein Zeitstempel T_i eines Prozesses p_i um die ID des Prozesses erweitert wird. Damit erhalten

die Zeitstempel die Form (T_i, i). Dann gilt: $(T_i, i) < (T_j, j)$ wenn entweder $T_i < T_j$ oder $T_i = T_j$ und $i < j$. Diese Reihenfolge erfolgt meist willkürlich, da die Prozess-IDs keiner Hierarchie untergeordnet sein müssen. Hierfür könnten z.B. die IP-Adressen einzelner Netzwerkteilnehmer verwendet werden. Der hieraus resultierende Zeitstempel ist nun eindeutig (vgl. [Coulouris et. al. 02], S. 463f).

Vektoruhren Eine Alternative zu Lamport-Zeitstempeln stellen die Vektoruhren dar. Diese wurden von Mattern und Fidge entwickelt, um aus der Tatsache, dass $L(e) < L(e')$ schließen zu können, dass $e \rightarrow e'$. Beim Einsatz von Vektoruhren in einem verteilten System mit N Prozessen erhält jeder Prozess einen Vektor V_i mit N Einträgen, wobei jeder Eintrag die einfache logische Zeit der einzelnen Prozesse beinhaltet. Um Vektoruhren zu erzeugen, müssen vier Regeln befolgt werden:

VC1 Zunächst ist $V_i[j] := 0$ für $i, j = 1, 2, ..., N$.

VC2 Unmittelbar bevor p_i einem Ereignis einen Zeitstempel zuweist, setzt er $V_i[i] := V_i[i] + 1$.

VC3 p_i gibt jeder versendeten Nachricht den Wert $t := V_i$ mit.

VC4 Wenn p_i einen Zeitstempel t in einer Nachricht empfängt, setzt er

$$V_i[j] := max(V_i[j], t[j]) \text{ für } j = 1, 2, ..., N.$$

Diese komponenetenweise Bildung des Maximalwerts für zwei Vektor-Zeitstempel wird auch als *merge*-Operation bezeichnet. Bevor dem Empfang der Nachricht ein Zeitstempel zugewiesen wird, wird **VC2** angewendet.

In Abbildung 3.4 werden die Vektor-Zeitstempel der Ereignisse aus Abbildung 3.2 gezeigt.

Mit dem Vektor-Zeitstempel-Verfahren lässt sich schnell festlegen in welcher Reihenfolge unterschiedliche Ereignisse auftreten:

$$V = V' \Leftrightarrow V[j] = V'[j] \text{ für } j = 1, 2, ..., N$$

$$V \leq V' \Leftrightarrow V[j] \leq V'[j] \text{ für } j = 1, 2, ..., N$$

$$V < V' \Leftrightarrow V \leq V' \wedge V \neq V'$$

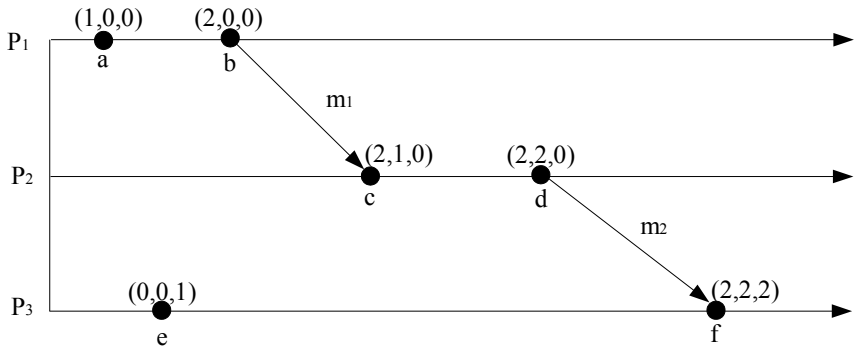


Abbildung 3.4: Ereignisabfolge mit Vektoruhren

Die Nebenläufigkeit zweier Ereignisse e und e' ist dann gegeben, wenn weder $V(e) \leq V(e')$ noch $V(e') \leq V(e)$ gilt.

Nachteil dieser Methode ist die hohe Speichernutzung, da für jeden Prozess p_i ein Eintrag in jedem Vektorzeitstempel vorhanden sein muss (vgl. [Coulouris et. al. 02], S. 464ff).

3.2 Globale Zustände

Ein Prozess p_i besitzt zu einem bestimmten Zeitpunkt t einen eindeutigen, rekonstruierbaren Zustand. Dieser kann in einem Verlauf festgehalten werden und ist somit jederzeit wieder abrufbar. Einen globalen Zustand eines verteilten Systems festzulegen ist hingegen nicht so einfach. Durch die Schwierigkeit der Zeitsynchronisierung der einzelnen Prozesse kann kein fester Zeitpunkt bestimmt werden, an dem ein solcher Zustand aufgenommen werden kann. Ein anderes Problem ist, dass eine Nachricht, die zwischen zwei Prozessen ausgetauscht wird, zu dem Zeitpunkt der Zustandsaufnahme zwar schon gesendet, jedoch noch nicht empfangen wurde. Der Inhalt dieser Nachricht wäre bei der Rekonstruktion des Zustands verloren. Es ist allerdings möglich, unter Einbeziehung der Kommunikationskanäle und der lokalen Zustände der einzelnen Prozesse einen globalen Zustand herzustellen.

Aufgaben, die mit Hilfe globaler Zustände gelöst werden können, sind:

- Verteilte Abfallsammlung (Distributed Garbage Collection) : Freigabe des

Speichers eines Objekts, von dem kein Verweis mehr in einem verteilten System existiert.

- Verteilte Verklemmungserkennung (Distributed Deadlock Recognition) : Erkennen einer Verklemmung, durch z.B. gegenseitiges Warten zweier Prozesse.
- Verteilte Terminierungserkennung (Distributed Termination Detection) : Erkennen der Beendigung eines verteilten Algorithmus, durch Passivität aller Prozesse. Diese Passivität kann auch nicht mehr aufgehoben werden.
- Verteiltes Debugging.

Um einen konsistenten, globalen Zustand Z zu garantieren, muss, falls ein Ereignis f enthalten ist, auch das Ereignis e in Z enthalten sein, sofern sich e zeitlich vor f ereignet hat. Diese zeitlich-vor-Beziehung wird mit Hilfe logischer Uhren (siehe Abschnitt 3.1.2) festgelegt. D.h. $f \in Z \wedge L(e) < L(f) \Rightarrow e \in Z$.

Abbildung 3.5 zeigt einen nach dieser Definition inkonsistenen und einen konsistenten Zustand (vgl. [Coulouris et. al. 02], S. 466ff).

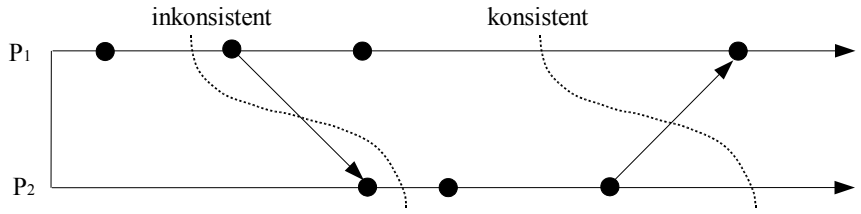


Abbildung 3.5: Inkonsistenter und konsistenter globaler Zustand

Schnappschuss-Algorithmus von Chandy und Lamport Chandy und Lamport verfolgen das Ziel, einen konsistenten, globalen Zustand eines verteilten Systems mit Hilfe eines virtuellen „Schnappschusses“ aufzuzeichnen. Der Algorithmus unterteilt das verteilte System, in dem er eingesetzt wird, in Prozesse und Kommunikationskanäle zwischen diesen Prozessen. Hierbei werden folgende Anforderungen gestellt:

- Kanäle und Prozesse sind fehlerfrei. Die Kommunikation muss zuverlässig sein, d.h. jede gesendete Nachricht kommt irgendwann bei ihrem Empfänger an.

- Die Kanäle sind unidirektional und agieren nach dem FIFO-Prinzip (First In, First Out).
- Alle Prozesse sind in beiden Richtungen miteinander verbunden.
- Ein Schnappschuss kann von jedem Prozess jederzeit initialisiert werden.
- Die Ausführung einzelner Aktionen der Prozesse wird durch den Schnappschuss nicht beinflusst.

Der eigentliche Algorithmus unterteilt sich in zwei Regeln:

1. Marker-Empfangsregel für Prozess p_i:

   Nachdem p_i eine Marker-Nachricht über Kanal c empfangen hat:
   if(p_i hat seinen Zustand noch nicht aufgezeichnet) then
   zeichnet er seinen Prozesszustand jetzt auf;
   zeichnet er den Zustand von c als leere Menge auf;
   aktiviert er die Aufzeichnung von Nachrichten, die über andere Eingangskanäle ankommen;
   else
   p_i zeichnet den Zustand von c als die Menge der Nachrichten auf, die er über c seit der letzten Zustandsaufzeichnung empfangen hat.
   end if

2. Marker-Senderegel für Prozess p_i:

   Nachdem p_i seinen Zustand aufgezeichnet hat, erfolgt für jeden Ausgangskanal c:
   p_i sendet eine Marker-Nachricht über c (bevor er andere Nachrichten über c sendet).

Es ist durchaus möglich, dass die über den „Schnappschuss"-Algorithmus aufgezeichneten Zustände nie wirklich gleichzeitig eingetreten sind (vgl. [Coulouris et. al. 02], S. 471ff).

3.3 Koordination und Übereinstimmung

Ein schwieriges Problem in verteilten System ist die Koordination oder die Übereinstimmung der einzelnen Prozesse. Eine zentrale Kontrollinstanz würde dieses

Problem beseitigen, allerdings wollte man sich eine zeitlang nicht auf eine solche verlassen und es sind Algorithmen entstanden, die diese Probleme auch ohne einen zentral agierenden Koordinator beseitigen könnnen.

Die Problematik der Koordination und Übereinstimmung wird in mehrere Teilbereiche untergliedert. Hierzu gehören der wechselseitige Ausschluss, die Wahl eines Anführers, die Festlegung der Nachrichtenreihenfolge und die Behebung von Übereinstimmungsproblemen, die alle in den folgenden Abschnitten erläutert werden.

3.3.1 Verteilter wechselseitiger Ausschluss

Das Problem des verteilten, wechselseitigen Ausschluss umfasst den Zugriff mehrerer Prozesse auf ein und dieselbe Ressource. Es wird auch als „kritischer-Abschnitt-Problem“ bezeichnet. Um hier die Konsistenz sicherzustellen, darf immer nur ein Prozess auf diese Ressource zugreifen oder nur ein Prozess den kritischen Abschnitt betreten.

Im Folgenden werden einige Algorithmen vorgestellt, die dieses Problem ohne eine zentrale Kontrollinstanz, z.B. in Form eines Servers, lösen. Dabei wird eine Abstimmung der einzelnen Prozesse untereinander angestrebt. An die Algorithmen werden bestimmte Anforderungen gestellt:

ME1 Sicherheitsaspekt. Höchstens ein Prozess kann den kritischen Abschnitt betreten.

ME2 Terminierungsaspekt. Anforderungen, in den kritischen Bereich einzutreten oder ihn zu verlassen, sind irgendwann erfolgreich.

ME3 Fairnessaspekt. Wenn ein Prozess vor einem anderen anfordert, in den kritischen Bereich einzutreten, wird der Eintritt in den kritischen Bereich in dieser Reihenfolge erteilt.

Um der letzten Anforderung gerecht zu werden, wird auf eindeutige, logische Uhren zurückgegriffen, die eine eindeutige Reihenfolge der Anfragen, den kritischen Bereich zu betreten, garantiert (vgl. [Coulouris et. al. 02], S. 492ff).

Wechselseitiger Ausschluss mit zentralem Server Eigentlich sollte bei dem wechselseitigen Ausschluss auf einen zentralen Server verzichtet werden. Es ist allerdings möglich, dass dieser von allen gleichberechtigten Prozessen gewählt wird (siehe Abschnitt 3.3.2). Aus diesem Grund wird auch diese Variante hier vorgestellt.

Um den kritischen Abschnitt zu betreten, sendet ein Prozess p_i seinen Eintrittswunsch an diesen Server. Dieser verwaltet eine Warteschlange, in die die

Anfragen eingehen und nach dem FIFO-Prinzip beantwortet werden, sobald der kritische Abschnitt wieder betreten werden kann. Nach dem Erhalt der Zusage des Servers kann der Prozess in den kritischen Abschnitt eintreten und teilt sein Verlassen dem Server mit, damit dieser dem nächsten Prozess in der Warteschlange den Eintritt zusichern kann.

Dieser Algorithmus arbeitet durch die zentrale Instanz sehr effizient, da für einen Eintritt nur drei Nachrichten versendet werden müssen, die Anfrage, die Erlaubnis und die Freigabemitteilung (vgl. [Coulouris et. al. 02], S. 494f).

Ring basierender Algorithmus Die einfachste Methode einen wechselseitigen Ausschluss ohne eine zentrale Komponente zu realisieren, ist die Anordnung der Prozesse in einem Ring, bei dem jeder Prozess einen Kommunikationskanal zu seinem benachbarten Prozess besitzt und die Übertragung von Nachrichten über diese Kanäle nur in einer Richtung stattfinden kann.

Um den kritischen Abschnitt zu betreten, benötigt ein Prozess ein bestimmtes Token, dass von den einzelnen Prozessen über den Ring gesendet wird. Dieses Token fungiert im Grunde wie ein Schlüssel, der den Zugang zu dem kritischen Abschnitt ermöglicht. Benötigt ein Prozess das Token, fordert er es über den Ring an, was bei einem System mit N Teilnehmern zwischen 0 und N Nachrichten erfordert. Erhält ein Prozess, der kein Interesse am Betreten des kritischen Abschnitts hat, das Token, sendet er es an seinen Nachbarprozess weiter.

Ein großer Nachteil dieses Algorithmus ist der hohe Verbrauch der Netzwerkbandbreite durch das ständige Senden von Nachrichten (vgl. [Coulouris et. al. 02], S. 495f).

Gegenseitiger Ausschluss von Ricart und Agrawala Ricart und Agrawala haben eine Lösung des Problems für ein Netzwerk mit Multicast-Funktionalität entwickelt. Voraussetzung dieses Algorithmus ist erneut eine eindeutige, logische Uhr (siehe Abschnitt 3.1.2). Der Algorithmus läuft wie folgt ab:

- Eintrittswunsch über Multicast an alle anderen Prozesse senden und auf deren Antworten warten.
- Nachdem eine Antwort von jedem anderen Prozess eingetroffen ist, kann der kritische Abschnitt betreten werden.
- Beim Eintreffen einer Anfrage zum Eintritt in den kritischen Abschnitt:
    1. falls selbst kein Eintrittswunsch vorliegt, Einverständnis zum Eintritt senden.

2. falls selbst Eintrittswunsch vorliegt und Zeitstempel der Anfrage kleiner als der der eigenen Anfrage, Einverständnis senden. Falls der eigene Eintrittswunsch vor der empfangenen Anfrage erfolgte, wird die Anfrage gepuffert.
3. falls selbst im kritischen Abschnitt, Anfragen puffern und nach Austritt Einverständnisse aller gepufferter Anfragen senden.

Abbildung 3.6 zeigt einen Ablauf des wechselseitigen Auschlusses nach Ricart und Agrawala. Bei diesem Beispiel liegt ein verteiltes System mit vier Prozessen (P_i mit $i = 1, .., 4$) vor. Prozess P_1 und P_4 äußern ihren Eintrittswunsch über Multicast bei allen anderen Prozessen. Dadurch, dass der Eintrittswunsch von P_1 durch die eindeutige logische Uhr (hier: $(1, 1)$) vor der von P_4 ($(1, 4)$) eingetreten ist ($L((1, 1)) < L((1, 4))$), muss P_4 zunächst P_1 den Vorrang lassen.

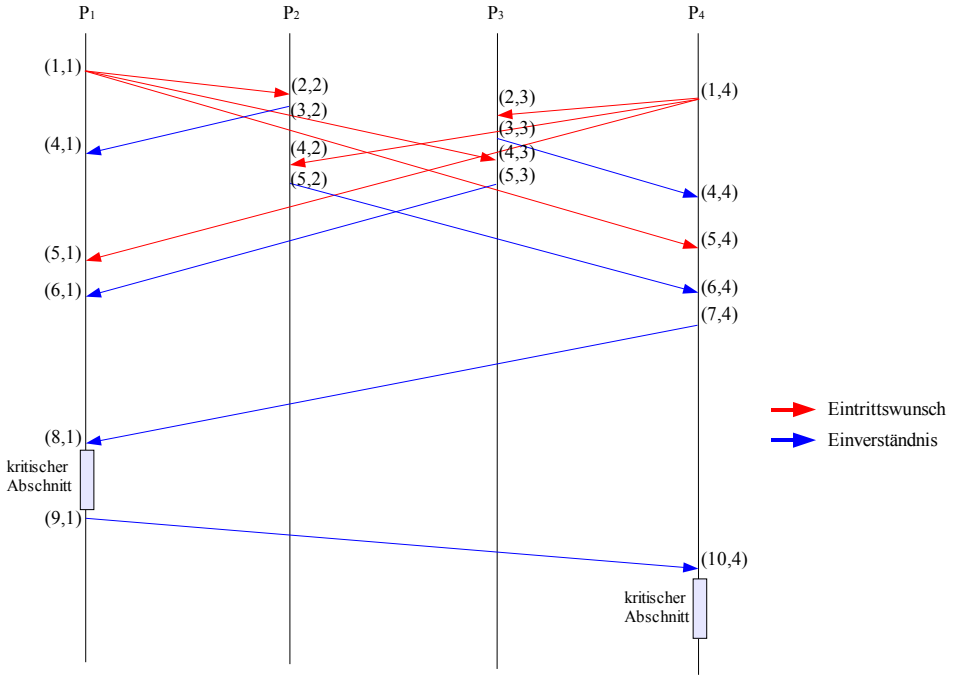


Abbildung 3.6: Wechselseitiger Ausschluss nach Ricart und Agrawala

Nachteil dieser Methode ist ebenfalls der hohe Bandbreitenverbrauch, da für einen Eintritt in den kritischen Abschnitt bei einem System mit N Teilnehmern, immer $2(N - 1)$ Nachrichten versendet werden müssen (vgl. [Coulouris et. al. 02], S. 496ff).

Wahl-Algorithmus von Maekawa Maekawa hat sich zum Ziel gesetzt, die bei den anderen Algorithmen zu hoch ausfallende Netzbandbreite möglichst gering zu halten. Dabei verfolgt er die Idee, dass ein Prozess nicht von allen eine Zustimmung braucht, sondern von einer Teilmenge der vorhandenen Prozesse gewählt wird, was ihn dazu berechtigt, den kritischen Abschnitt zu betreten.

Um beide oben gesetzten Ziele zu verfolgen, unterteilt Maekawa ein Netzwerk in „Wählergruppen“ ein. Einzige Voraussetzung, die er an diese Teilmengen stellt, ist, dass sich die Teilmengen zweier beliebiger Prozesse überlappen. D.h. jeder Prozess $p_i (i = 1, 2, ..., N)$ wird einer Wählergruppe V_i zugeordnet, wobei

$$V_i \subseteq \{p_1, p_2, ..., p_N\}.$$

Für die Mengen V_i und alle $i, j = 1, 2, ..., N$ gelten folgende Regeln:

- $p_i \in V_i$
- $V_i \cap V_j \neq \oslash$, d.h. es gibt mindestens ein gemeinsames Element von zwei beliebigen Teilmengen.
- $|V_i| = K$, d.h. alle Mengen müssen die gleiche Anzahl Prozesse beinhalten
- Jeder Prozess p_i ist in M der Wählermengen V_i enthalten.

Die optimale Lösung dieses Algorithmus tritt, laut Maekawa, dann ein, wenn K so klein wie möglich gewählt wird, so dass $K \approx \sqrt{N}$ und $M = K$ ist.

Der Ablauf des Algorithmus sieht wie folgt aus:

- Jeder Prozess erhält eine „gewählt“-Markierung, die anzeigt, ob er schon einem Prozess eine Zusage zum Betreten des kritischen Abschnitts gesendet hat oder nicht. Bei der Initialisierung wird dieser Wert auf *false* gesetzt.
- Bei Eintrittswunsch von p_i sendet dieser eine Anfrage an alle Gruppenmitglieder $(V_i - p_i)$ seiner Wahlmenge und wartet bis er von allen eine Antwort erhält. Nachdem alle Antworten bei ihm eingetroffen sind, kann er den kritischen Abschnitt betreten.
- Bei Erhalt einer Anfrage prüft der Empfänger, ob er sich im kritischen Abschnitt befindet oder ob er jemandem bereits eine Zusage gesendet hat.
  1. Ist dies der Fall, wird die Anfrage in einer Warteschlange eingereiht.
  2. Ansonsten sendet er eine Zusage und setzt seine „gewählt“-Markierung auf *true*.
- Nach dem Verlassen des kritischen Abschnitts teilt p_i dies allen seinen Gruppenmitgliedern mit.

- Bei Erhalt einer Nachricht, die die Freigabe des kritischen Abschnitts beinhaltet, wird die Warteschlange geprüft:

  1. Ist diese nicht leer, bekommt der erste dieser Schlange eine Zusage und die „gewählt"-Markierung wird auf *true* gesetzt.
  2. Befindet sich keine Anfrage in der Warteschlange, wird die „gewählt"-Markierung auf *false* gesetzt.

Abbildung 3.7 zeigt ein Beispiel für den Ablauf des Verfahrens von Maekawa. Bei Betrachtung dieses Beispiels wird deutlich, dass für den Eintritt in den kritischen Bereich $2(K-1)$ Nachrichten und für den Austritt $K-1$ Nachrichten benötigt werden. Tritt der Optimalfall ein, dies ist bei $N > 4$ der Fall, beschränkt sich die Bandbreite für Ein- und Austritt des kritischen Bereichs auf $3\sqrt{N}$ Nachrichten.

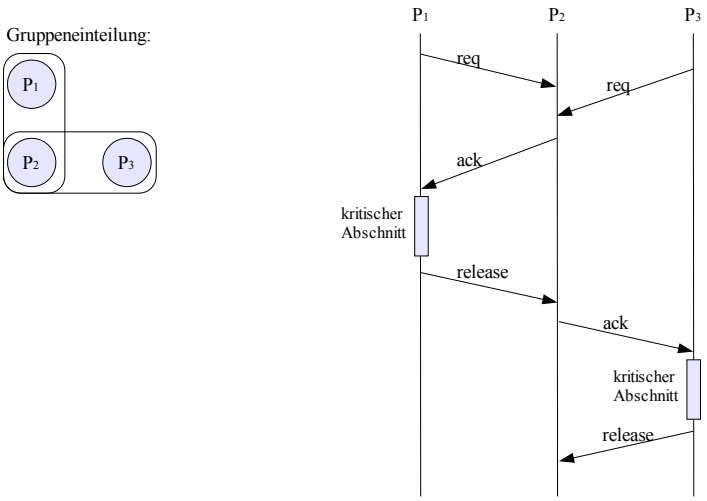


Abbildung 3.7: Wechselseitiger Ausschluss nach Maekawa

Bei diesem Verfahren besteht allerdings Verklemmungsgefahr. Abbildung 3.8 zeigt ein solches Szenario, bei dem alle vorhandenen Prozesse den kritischen Abschnitt gleichzeitig betreten möchten. Dadurch, dass sie nur über eine Stimme verfügen, können sie jeweils nur einer Anfrage zustimmen. Um den kritischen Abschnitt zu betreten sind aber zwei Antworten nötig, auf die jeder Prozess vergebens wartet. Mit Hilfe eindeutiger, logischer Uhren (Abschnitt 3.1.2), lässt sich die Verklemmungsgefahr beseitigen (vgl. [Coulouris et. al. 02], S. 498ff).

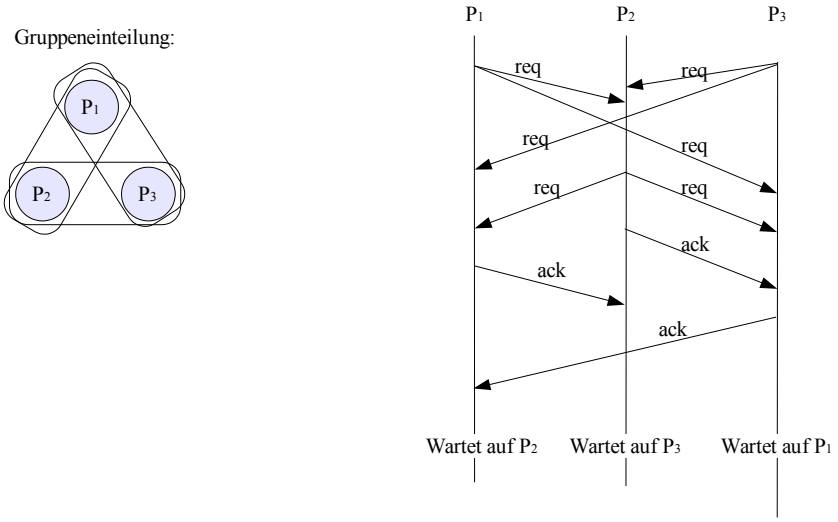


Abbildung 3.8: Wechselseitiger Ausschluss nach Maekawa mit Verklemmung

3.3.2 Wahl eines Anführers

Die meisten der vorangegangenen Algorithmen haben auf eine zentrale Instanz verzichtet. Bei einigen ist dies jedoch unumgänglich. Um aus einem verteilten System, bestehend aus gleichberechtigten Prozessen, einen bestimmten Prozess zu bestimmen, der die Rolle dieser zentralen Instanz übernimmt, werden Wahl-Algorithmen verwendet.

Eine wichtige Eigenschaft solcher Wahl-Verfahren ist, dass jeder Prozess p_i jederzeit eine Wahl initiieren kann. Die Initiierung einer neuen Wahl kann beispielsweise dann erfolgen, wenn ein Prozess merkt, dass der derzeit gewählte Prozess nicht mehr verfügbar ist (vgl. [Coulouris et. al. 02], S. 501f).

Ringbasierte Wahl Chang und Roberts haben eine Methode entwickelt, um einen Anführer in einer Ringstruktur festzulegen. Dabei wird der Prozess mit der höchsten ID gewählt. Das Verfahren läuft wie folgt ab:

- Um die Wahl zu initiieren, wird eine Wahlnachricht mit der eigenen ID an den nächsten Prozess gesendet. Zusätzlich merkt sich der Prozess, dass er bereits Teilnehmer einer Wahl ist.
- Bei Erhalt einer Wahlnachricht überprüft der Empfänger seine ID mit der der Nachricht.

1. Ist die eigene ID kleiner als die empfangene, sendet er die Nachricht an den nächsten Prozess weiter und setzt sich als Teilnehmer.
2. Ist die eigene ID größer und der Empfänger noch kein Teilnehmer einer Wahl, trägt er seine eigene ID in die Nachricht ein, sendet sie weiter und setzt sich als Teilnehmer. Ansonsten wird die Nachricht verworfen.
3. Ist die empfange ID gleich der eigenen, dann ist der Empfänger der neue Anführer. Dieser setzt sich als Nicht-Teilnehmer und sendet eine Wahlergebnisnachricht mit seiner eigenen ID an den nächsten Prozess.

- Bei Empfang einer Wahlergebnisnachricht merkt sich der Empfänger die ID des neuen Anführers, markiert sich selbst als Nicht-Teilnehmer und sendet die Nachricht an den nächsten Prozess weiter, falls er nicht selbst Anführer ist.

Abbildung 3.9 zeigt einen möglichen Ablauf dieses Algorithmus. Die Initialisierung wurde in diesem Beispiel von Prozess „19“ gestartet.

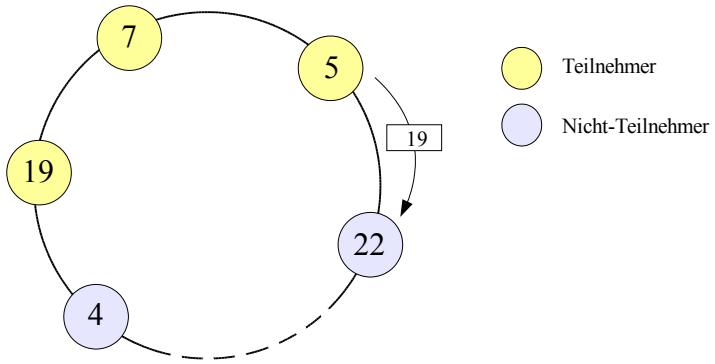


Abbildung 3.9: Wahlverfahren im Ring

Wird dieses Verfahren von nur einem Prozess initiiert, werden im schlimmsten Fall $3N-1$ Nachrichten benötigt, um den neuen Anführer bestimmen zu können. Dieser Fall tritt genau dann ein, wenn der zukünftige Anführer der Prozess ist, der in der Ringreihenfolge vor dem Initiator kommt. Dann benötigt die Wahlnachricht $N-1$ Schritte, um zu ihm zu gelangen, und noch mal N Schritte bis dessen Wahl bestätigt werden kann. Um das Ergebnis um den Ring zu schicken, werden weitere N Nachrichten verwendet. Im besten Fall startet der zukünftige

Anführer die Wahl selbst, was $2N$ Nachrichten benötigt. Die Markierung als Teilnehmer, wodurch zusätzliche „Zweitwahlen“ vermieden werden, erspart weitere Netzwerkbandbreite (vgl. [Coulouris et. al. 02], S. 502f).

Alternatives Ringverfahren Um den *worst case* bei dem vorangegangenen Algorithmus $(3N - 1)$ zu verbessern, kann eine alternative Variante zur Wahl eines Anführers in einer Ringstruktur verwendet werden:

- Der Initiator einer Wahl sendet eine Wahlnachricht mit einer ID-Liste, in der zunächst nur seine eigene Kennung enthalten ist, an den Ring.
- Bei Empfang einer Wahlnachricht trägt der Prozess seine ID in die Liste ein und sendet sie weiter.
- Kommt die Liste aller IDs an den Initiator zurück, bestimmt dieser daraus das Maximum und sendet eine Wahlergebnisnachricht.

Mit diesem Verfahren wird der schlimmste Fall auf $2N$ Nachrichten reduziert. Eine Wahl mit dieser Methode benötigt immer $2N$ Nachrichten. Nachteil dieses Verfahrens ist allerdings der große Speicheraufwand der Nachricht. Der Einsatz einer Teilnahmemarkierung, würde bei dieser Methode ebenfalls zu einer Erniedrigung der Bandbreite führen, da es ausreicht einen Wahlgang zu beenden (vgl. [Oechsle02], Seite 16).

Bully-Algorithmus Beim Bully-Algorithmus von Garcia-Molina wird vorausgesetzt, dass jeder Prozess eines verteilten Systems die Prozess-IDs der anderen Prozesse kennt. Ein Wahlgang mit diesem Verfahren läuft wie folgt ab:

1. Merkt ein Prozess, dass der derzeitige Anführer ausgefallen ist, startet er eine Wahl, sofern er nicht die nunmehr höchste Kennung besitzt (bei höchster Kennung: Schritt 5.).
2. Um eine Wahl zu starten, werden Wahlnachrichten an alle Prozesse mit einer höheren ID gesendet und auf dessen Antwort gewartet.
3. Trifft nach einem Timeout keine Antwort von einem Prozess ein, gilt dieser als ausgefallen.
4. Beim Eintreffen einer Wahlnachricht initiiert der Empfänger eine eigene Wahl (1.).
5. Stellt ein Prozess fest, dass er die höchste ID im System hat, z.B. dadurch, dass er keine Antwort von einem Prozess mit höherer ID erhalten hat, sendet er eine Koordinatornachricht an alle anderen Prozesse um ihnen dies mitzuteilen.

Das Beispiel in Abbildung 3.10 wird durch P_1 initiiert, der zuerst bemerkt hat, dass P_4 ausgefallen ist. Nachdem P_3 festgestellt hat, dass er die höchste Kennung besitzt, kann er dies den anderen Prozessen mitteilen.

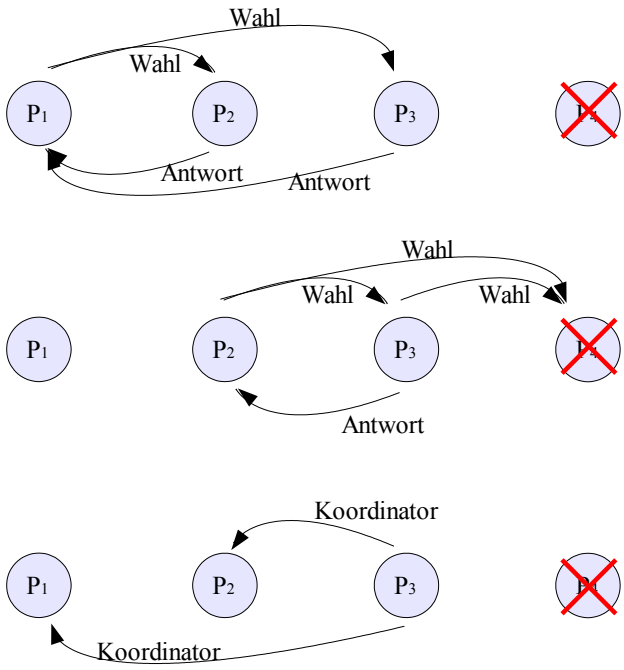


Abbildung 3.10: Wahl durch Bully-Algorithmus

Im besten Fall benötigt diese Methode $N - 2$ Nachrichten, nämlich genau dann, wenn der Prozess mit der zweithöchsten Kennung feststellt, dass der Koordinator ausgefallen ist. Dann kann dieser unmittelbar allen anderen Prozessen mitteilen, dass er der neue Koordinator ist. Im schlimmsten Fall werden $O(N^2)$ Nachrichten benötigt (vgl. [Coulouris et. al. 02], S. 503ff).

3.3.3 Nachrichtenreihenfolge

In Abschnitt 3.1.2 wurde gezeigt, wie wichtig es ist, zu wissen, in welcher Reihenfolge bestimmt Ereignisse auftreten. Ähnlich verhält es sich mit Nachrichten eines verteilten Systems und der Reihenfolge, in der sie von den verschiedenen Prozessen bearbeitet werden. Dieses Problem tritt vor allem bei Netzwerken

auf, in denen mit Multicast oder einem simulierten Multicast gearbeitet wird. Dort kann die eigentliche Reihenfolge mehrerer Nachrichten durch längere Verzögerungszeiten bei den einzelnen Prozessen vertauscht werden. Es gibt mehrere Möglichkeiten, die Reihenfolge dieser Nachrichten zu definieren:

FIFO Wenn ein Prozess p_i zuerst Nachricht m versendet hat und anschließend m', dann wird auch die Nachricht m vor der Nachricht m' bei den Empfängern ausgeliefert.

kausal Eine Nachrichtenreihenfolge ist kausal, wenn der Emfang von Nachricht m das Senden der Nachricht m' auslöst und diese Nachrichten auch in dieser Reihenfolge ausgeliefert werden.

total Unter der totalen Reihenfolge wird eine einheitliche Reihenfolge bei allen Prozessen verstanden. Diese muss aber weder FIFO noch kausal sein.

Liegt eine kausale Reihenfolge vor, impliziert dies auch die FIFO-Reihenfolge. Zusätzlich zu den drei Formen gibt es die Hybridformen:

- FIFO-total
- kausal-total

Abbildung 3.11 zeigt ein Beispiel zu jeder einfachen Form (vgl. [Coulouris et. al. 02], S. 513ff).

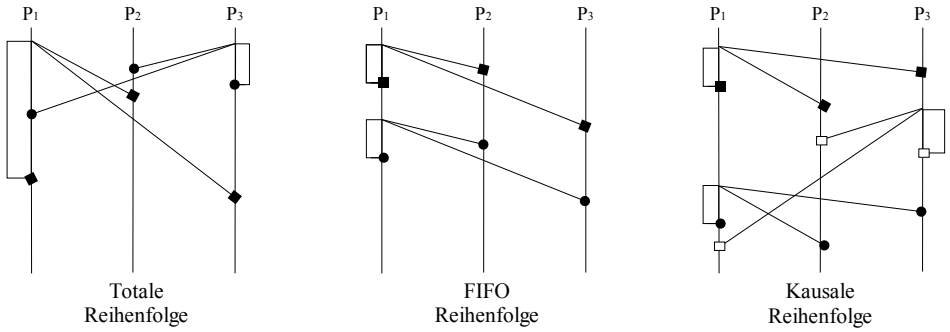


Abbildung 3.11: Totale, FIFO und kausale Reihenfolge

Im Folgenden werden einige Lösungen für die einzelnen Formen vorgestellt.

Totale Reihenfolge mit Primärempfänger Die einfachste Methode eine totale Reihenfolge zu gewinnen, ist einen Primärempfänger zu bestimmen, der alle Nachrichten abfängt und an die anderen Prozesse weiterleitet. Der Primärempfänger könnte vorher aus den Prozessen gewählt worden sein (siehe Abschnitt 3.3.2). Einzige Voraussetzung dieses Algorithmus ist, dass die Kommunikationskanäle vom Primärempfänger zu den anderen Prozessen FIFO sind. Abbildung 3.12 zeigt den Ablauf dieses Verfahrens.

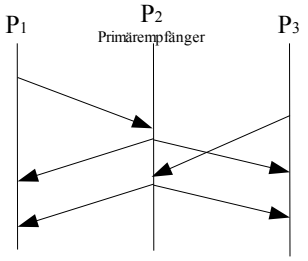


Abbildung 3.12: Totale Reihenfolge durch Primärempfänger

Nachteil dieser Methode ist natürlich die Abhängigkeit von einer zentralen Komponente und ein möglicher Nachrichtenverlust bei deren Ausfall sowie die Verzögerungszeit, die durch den Umweg über den Primärempfänger auftritt (vgl. [Oechsle02], S. 20).

Totale Reihenfolge mit Sequenzer Eine Variante des Verfahrens mit Primärempfänger ist die mit dem Einsatz eines sogenannten Sequenzers. Dieser kann ebenfalls aus den Prozessen gewählt werden. Der Algorithmus läuft wie folgt ab:

1. Verhalten eines normalen Prozesses:
   - Alle Prozesse besitzen einen Zähler c, der mit 0 initialisiert wird.
   - Eine Nachricht m wird mit einer eindeutigen Nachrichtenkennung i in der Form $< m, i >$ gesendet.
   - Bei Empfang einer Nachricht wird diese in eine Rückhalte-Warteschlange gelegt.
   - Bei Empfang einer Sortiernachricht des Sequenzers, die eine Ordnungsnummer S und eine Nachrichtenkennung i der Form $< S, i >$ enthält, wird solange gewartet bis die Nachricht mit der entsprechenden Kennung in der Warteschlange eingegangen ist und die Ordnungsnummer $S = c + 1$ ist. Dann wird die entsprechende Nachricht aus der Rückhalte-Schlange entfernt und der Zähler $c := S$ gesetzt.

2. Verhalten des Sequenzers:

   - Der Sequenzer besitzt einen eigenen Zähler S, der mit 0 initialisiert wird.
   - Bei Erhalt einer Nachricht mit der Kennung $< m, i >$ wird eine Sortiernachricht mit der Kennung der empfangenen Nachricht i sowie dem Wert des aktuellen Zählers $S < S, i >$ gesendet. Anschließend wird der Zähler erhöht $S := S + 1$.

Aber auch dieser Algorithmus hat den Nachteil, dass er auf den Sequenzer und dessen Zuverlässigkeit angewiesen ist (vgl. [Coulouris et. al. 02], S. 516ff).

Totale Reihenfolge durch Zeitsteuerung Ein dezentraler Ansatz für eine totale Reihenfolge, ist der, eine Zeitsteuerung zu verwenden. Voraussetzung dieses Verfahrens ist, dass ein Prozess die Zeit t abschätzen kann, d.h. wie lange eine Nachricht höchstens braucht, um beim Empfänger anzukommen. Unter dieser Voraussetzung verläuft der Algorithmus wie folgt:

- Eine Nachricht m wird mit der geschätzten Zeit t, die die Nachricht zum Empfänger benötigt, in der Form $< m, t >$ gesendet.
- Bei Erhalt einer Nachricht $< m, t >$ wird diese gepuffert.
- Eine Nachricht wird aus dem Puffer entfernt und ausgeliefert, sobald der Zeitpunkt t erreicht ist.
- Wird eine Nachricht empfangen, deren geschätzte Ankunftzeit bereits abgelaufen ist, wird diese verworfen.

Nachteil dieser Methode ist zum einen, dass die Nachrichten erst nach Ablauf der geschätzen Zeitspanne ausgeliefert und verarbeitet werden können. Wird diese Zeitspanne zu hoch gesetzt, verlangsamt sich der Algorithmus. Wird sie zu niedrig gewählt, besteht die Gefahr, dass mehrere Nachrichten gar nicht erst ausgeliefert werden (vgl. [Oechsle02], S. 20).

Totale Reihenfolge durch Folgenummern Die Methode, eine totale Reihenfolge durch Folge- oder Ordnungsnummern zu erhalten, ähnelt ein wenig dem zentralen Verfahren über einen Sequenzer. In diesem Fall wird die Ornungsnummer einer Nachricht aber nicht von einer zentralen Instanz festgelegt, sondern wird von allen Prozessen gemeinsam bestimmt. Der Ablauf dieser Einigung sieht folgendermaßen aus:

- Eine Nachricht m wird mit einer eindeutigen ID i und der momentanen Folgenummer n des sendenden Prozesses (ähnlich einer eindeutigen, logischen Uhr) $< m, i, n >$ verschickt.
- Bei Empfang einer solchen Nachricht setzt der Empfänger p_i seine Folgenummer $n_i := max(n_i, n) + 1$ und antwortet dem Sender mit einer Nachricht, die die Kennung der empfangenen Nachricht i und seine neue Folgenummer n_i beinhaltet $(< i, n_i >)$. Die empfangene Nachricht wird mit der neuen Folgenummer in eine Rückhalte-Warteschlange einsortiert, in der die Nachricht mit der kleinsten Folgenummer vorne steht.
- Sind alle Antworten einer Nachricht eingetroffen, wird dieser die größte empfangene Folgenummer n übergeben und eine Zuordnungsnachricht mit der Nachrichtenkennung i und der endgültigen Folgenummer n wird an alle Prozesse gesendet.
- Empfängt ein Prozess diese Zuornungsnachricht, ändert er gegebenenfalls die Folgenummer der Nachricht und anschließend seine eigene momentane Folgenummer n_i auf $n_i := max(n_i, n)$. Sollte sich die Nachricht, deren Folgenummer nun feststeht, am Anfang der Rückhalte-Schlange befinden, wird sie ans Ende der Auslieferungsschlange eingeordnet.

Abbildung 3.13 zeigt ein Beispiel, bei dem zwei Nachrichten m_1 und m_2 in eine totale Reihenfolge sortiert werden.

Nachteil dieser Methode ist die häufige Kommunikation. Bevor eine Nachricht ausgeliefert werden kann, müssen jeweils drei Nachrichten zwischen dem Sender und den einzelnen Empfängern ausgetauscht werden $(3(N-1))$. Dieser Algorithmus beschränkt sich auf die totale Reihenfolge, die entstandene Reihenfolge muss weder FIFO noch kausal sein. Einen praktischen Einsatz dieses Verfahrens findet sich im ISIS-System (vgl. [Coulouris et. al. 02], S. 516f).

Kausale Reihenfolge durch Vektoruhren In Abschnitt 3.1.2 wurden bereits die Vektoruhren eingeführt, um festzustellen, ob ein Ereignis vor einem anderen Ereignis ausgelöst wurde oder es sogar ein anderes Ereignis zur Folge hat. Mit einer ähnlichen Strategie lässt sich auch eine kausale Reihenfolge von Nachrichten ermitteln. Jeder Prozess p_i eines verteilten Systems mit N Mitgliedern besitzt einen Vektor $V_i[j]$ mit $j = 1, 2, ..., N$, in dem für jeden Prozess p_j ein Eintrag dessen logischer Zeit existiert. Die Arbeitsweise dieses Verfahrens ist:

- Die Vektoren werden mit $V_i[j] := 0 (j = 1, 2, ..., N)$ initialisiert.
- Sendet Prozess p_i eine Nachricht m, erhöht er zuvor seinen eigenen Zeitstempel $V_i[i] = V_i[i] + 1$ und sendet diesen Vektor mit der Nachricht $< V_i, m >$.

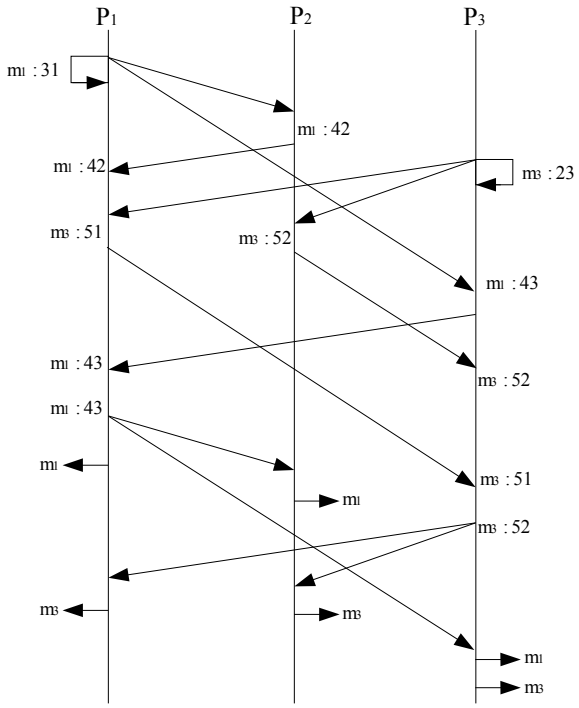


Abbildung 3.13: Totale Reihenfolge durch Folgenummern

- Empfängt p_i die Nachricht $< V_j, m >$ von p_j, wird diese zunächst in eine Rückhalte-Warteschlange eingefügt, bis $V_j[j] = V_i[j] + 1$ und $V_j[k] \leq V_i[k](k \neq j)$.
- Nach dem Entfernen dieser Nachricht aus der Warteschlange wird $V_i[j] := V_i[j] + 1$ gesetzt.

Ein möglicher Ablauf dieser Methode wird in Abbildung 3.14 dargestellt. Der Algorithmus kann feststellen, dass sich die Nachrichten m_1 und m_2 bei P_3 vertauscht haben und bringt sie in die richtige Reihenfolge.

Es besteht die Möglichkeit, dieses Verfahren um einen Sequenzer, wie er für eine totale Reihenfolge eingesetzt wird, zu erweitern, um eine total-kausale Reihenfolge zu erhalten (vgl. [Coulouris et. al. 02], S. 519ff).

3.3.4 Übereinstimmungsprobleme

Bisher wurde bei allen vorgestellten Algorithmen ein korrektes Verhalten der Prozesse vorausgesetzt. Dieser Abschnitt beschäftigt sich mit der Problematik, dass

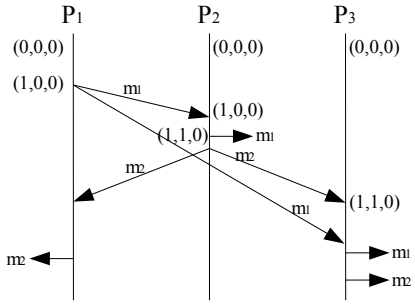


Abbildung 3.14: Kausale Reihenfolge durch Vektoruhren

Prozesse untereinander einen Konsens erreichen müssen, auch wenn einer oder mehrere von ihnen fehlerhaft arbeiten oder gar ausfallen. Dabei werden drei Probleme vorgestellt:

1. Konsens,
2. interaktiver Konsens und
3. das Problem der byzantinischen Generäle.

Die einzelnen Probleme werden im Folgenden behandelt, wobei die Lösungsverfahren zunächst als Blackbox vorliegen.

Konsens In einem System mit N Prozessen schlägt jeder Prozess p_i mit

$$i = 1, 2, ..., N$$

einen Wert d_i aus einer Liste möglicher Werte $d_i \in \{V_1, ..., V_M\}$ vor. Am Ende des Verfahrens haben sich alle korrekt arbeitenden Prozesse auf denselben Wert V_j geeinigt. Wenn alle Prozesse denselben Wert vorgeschlagen haben, müssen sie sich auch auf diesen geeinigt haben.

interaktiver Konsens Der interaktive Konsens ist eine Erweiterung des Konsens-Problems. Wie vorher auch schlägt jeder Prozess p_i einen Wert d_i aus einer Reihe möglicher Werte vor. Die einzelnen Vorschläge jedes Prozesses werden jeweils in einem Vektor D_i aufgenommen. Ziel des Verfahrens ist es, dass alle korrekt arbeitenden Prozesse auch denselben Entscheidungsvektor besitzen. Daraus folgt, dass, wenn ein korrekter Prozess p_i einen Wert d_i vorschlägt, so befindet sich dieser Wert im Vektor aller korrekten Prozesse ($V_j[i] = d_i$).

byzantinische Generäle Das Problem der byzantinischen Generäle wird durch ein Szenario beschrieben, in welchem es mehrere Generäle und einen befehlshabenden Kommandeur gibt. Dieser erteilt allen Generälen entweder den Befehl „Angriff“ oder „Rückzug“. Die Generäle müssen sich nun auf einen dieser beiden Befehle einigen. Dabei soll berücksichtigt werden, dass es Generäle gibt, die sich dem Kommandeur gegenüber nicht loyal verhalten und vorgeben, sie hätten einen anderen Befehl erhalten. Es könnte aber auch der Fall sein, dass der Kommandeur sich als nicht loyal entpuppt und den einzelnen Generälen unterschiedliche Befehle erteilt. Ziel ist die Einigung aller loyalen Generäle auf einen Befehl. Ist der Kommandeur loyal, müssen die Generäle auch den von ihm vorgeschlagenen Befehl ausführen.

Bisher wurde noch kein Lösungsverfahren für eines der drei Probleme vorgestellt. Es würde allerdings ausreichen, nur die Lösung eines dieser Probleme zu kennen, da sich die Lösungen der anderen aus dieser ableiten lassen:

Konsens $\Rightarrow$ byzantinische Generäle

- Der Kommandeur p_j sendet seinen Befehl d_j an alle Generäle.
- Diese einigen sich über den Konsens auf einen der erhaltenen Befehle, wobei p_j nicht korrekt arbeiten muss.

byzantinische Generäle $\Rightarrow$ interaktiver Konsens Das Problem der byzantinischen Generäle wird N-mal gelöst, wobei jeder Prozess p_i einmal der befehlshabende Kommandeur ist.

interaktiver Konsens $\Rightarrow$ Konsens Auf den durch den interaktiven Konsens entstanden Entscheidungsvektor wendet jeder Prozess p_i die selbe Funktion (z.B. $max()$, $min()$, $avg()$) an, um so einen gemeinsamen Wert zu erhalten.

Konsens Dolev und Strong haben einen Algorithmus entwickelt, der das Problem des Konsens behebt. Dieser wird in $f + 1$ Runden durchgeführt, wobei höchstens f Prozesse abstürzen dürfen. Jeder Prozess p_i besitzt einen Vektor von Mengen V_i, in denen alle empfangenen, vorgeschlagenen Werte d_j der anderen Prozesse p_j enthalten sind. Der Algorithmus läuft wie folgt ab:

- Wird p_i initialisiert, setzt er die Menge $V_i[0] = \{\}$ und $V_i[1] = \{d_i\}$.
- Solange $1 \leq r \leq f + 1$:
  - p_i sendet alle noch nicht gesendeten Werte $V_i[r] - V_i[r - 1]$ an alle anderen Prozesse und

  - trägt alle bisherigen Werte an die nächste Stelle des Vektors $V_i[r+1] = V_i[r]$.
  - Bei Erhalt einer Nachricht mit V_j von p_j, wird die erhaltene Menge mit der nächsten eigenen vereint $V_i[r+1] := V_i[r+1] \cup V_j$.
- Nach Ablauf von $f+1$ Runden wird $d_i = min(V_i[f+1])$ gesetzt.

Ein Vorteil dieses Algorithmus ist das Erkennen von Ausfällen, wodurch allerdings eine hohe Netzauslastung aufgewendet werden muss, da der Algorithmus $f+1$ mal durch geführt wird (vgl. [Coulouris et. al. 02], S. 527f).

Problem der byzantinischen Generäle Eine Lösung des Problems der byzantinischen Generäle sollte berücksichtigen, dass einige Generäle Verräter sind. In der Computerwelt würde das bedeuten, dass einige Prozesse falsche oder überhaupt keine Nachrichten ausliefern. Der folgende Algorithmus löst dieses Problem mit einer Einschränkung. Ein verteiltes System mit f fehlerhaften Prozessen muss mindestens aus $N \geq 3f+1$ Prozessen bestehen, um eine Einigung zu erzielen. Abbildung 3.15 zeigt, dass es unmöglich ist, zwischen P_2 und P_3 eine Übereinstimmung zu erreichen, da beide einen unterschiedlichen Wert vorgeschlagen haben, auf den sie sich einigen müssten.

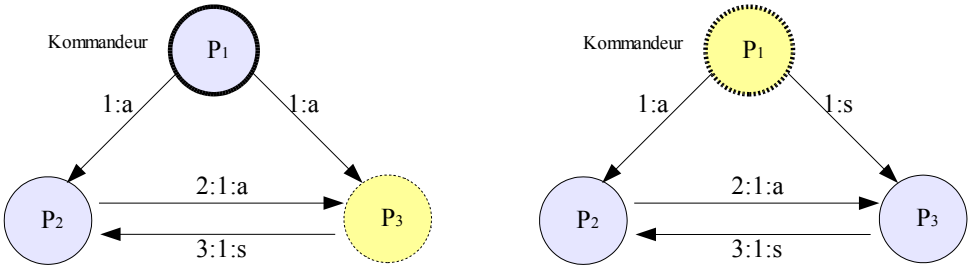


Abbildung 3.15: Drei byzantinische Generäle

Aus der Abbildung wird ebenfalls die Arbeitsweise des Algorithmus ersichtlich:

- Der Kommandeur sendet seinen Befehl an alle Generäle.
- Jeder General p_i sendet seinen vom Kommandeur erhaltenen Befehl d_i an alle anderen Generäle.
- Ein General führt den am häufigsten erhaltenen Befehl aus oder alle Generäle führen einen Standardbefehl $\perp$ aus, falls keine Mehrheit erreicht wurde.

Abbildung 3.16 zeigt zwei Beispiele mit vier Generälen, von denen einer jeweils ein Verräter ist. In beiden Fällen einigen sich die Generäle auf einen Befehl. Im ersten Beispiel können sich die beiden fehlerlosen Prozesse P_2 und P_4 auf den Befehl u einigen. Im zweiten Beispiel ist der Kommandeur der fehlerhafte Prozess und sendet jedem General einen anderen Befehl, woraus diese keine Mehrheit gewinnen können und den Standardbefehl $\perp$ ausführen.

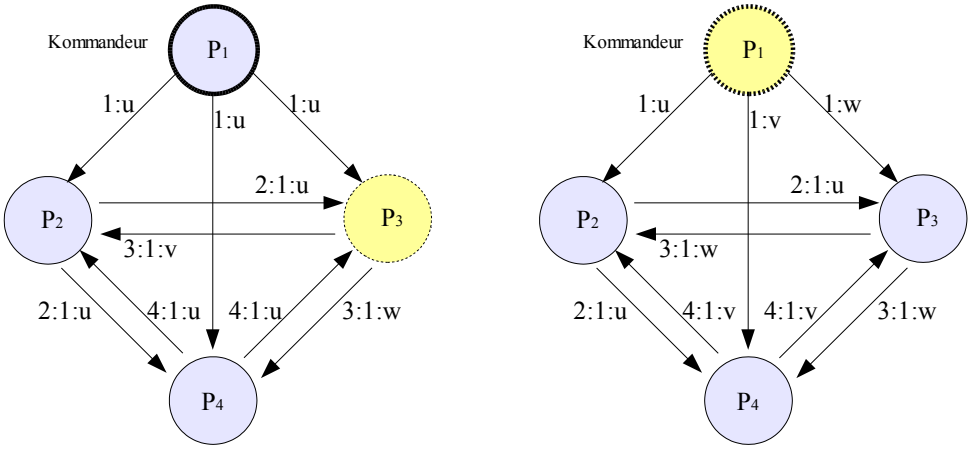


Abbildung 3.16: Vier byzantinische Generäle

Wird davon ausgegangen, dass auch wirklich nur f Prozesse fehlerhaft sind, kann mit Hilfe dieses Algorithmus der oder die Veräter enttarnt werden. Eine Möglichkeit, um dieses Verfahren weiter zu verbessern und sogar für drei Prozesse abzusichern, wäre eine Verwendung von Signaturen, wodurch allerdings die Bandbreite auf $O(N^2)$ steigen würde (vgl. [Coulouris et. al. 02], S. 528ff).

4 Überblick und Grobkonzept

4.1 Programmkomponenten

Das in dieser Arbeit entstandene Programm nennt sich DisASTer, was für *Distributed Algorithms Simulation Terrain* steht. Das Programm besteht aus zwei Teilen:

1. einer Klassenbibliothek, die Klassen enthält, um verteilte Algorithmen implementieren zu können. Diese Bibliothek ist der eigentliche Baukasten, aus dem ein Entwickler Klassen und Methoden verwenden kann, um verteilte Algorithmen zu programmieren.

2. einem Programm mit einer grafischen Oberfläche, in der die mit der Bibliothek erstellten Algorithmen simuliert, visualisiert und getestet werden können.

4.2 Eigenschaften verteilter Algorithmen

Bevor die Arbeitsweise von DisASTer beschrieben wird, werden die Gründe aufgeführt, die zu der jetzigen Architektur von DisASTer geführt haben. Ziel der Entwicklung von DisASTer war es, möglichst viele verteilte Algorithmen realisieren zu können. Um diesem Ziel gerecht zu werden, wurden zunächst die Eigenschaften zusammengetragen, die alle verteilten Algorithmen besitzen. Da verteilte Algorithmen in verteilten Systemen, wie Netzwerken, eingesetzt werden, lassen sich schnell einige triviale Eigenschaften festhalten, die mit dem Aufbau eines verteilten Systems eng verbunden sind:

1. Ein verteiltes System besteht aus mehreren Teilnehmern oder Prozessen, die bestimmte Zustände besitzen oder unterschiedliche Rollen annehmen können.

2. Diese Prozesse sind über Kommunikationskanäle miteinander verbunden.

3. Über diese Kanäle tauschen die Prozesse Nachrichten untereinander aus, welche

4. bestimmte Aktionen und Ereignisse bei ihren Empfängern auslösen. Der Empfang einer Nachricht könnte z.B. eine Zustandsänderung bewirken.

Diese einfache Ablaufbeschreibung umfasst nahezu alle verteilten Algorithmen, womit das Ziel, eine allgemeine Beschreibung verteilter Algorithmen zu finden, erreicht ist. Der nächste Schritt befasst sich mit der Umsetzung dieser Erkenntnisse in DisASTer.

Hierzu wird zunächst auf das verteilte System eingegangen, in dem ein Algorithmus seine Aufgaben erledigt. Ein solches System besitzt eine Topologie, die aus den einzelnen Prozessen und deren Verbindungen besteht. Dieselben Elemente finden sich auch in der Klassenbibliothek wieder. Die für den Benutzer des Baukastens wichtigste Klasse ist die abstrakte Klasse `Agent`. Ein Agent stellt einen Teilnehmer eines verteilten Systems dar. Hauptaufgabe des Anwenders ist, die Klasse `Agent` zu erweitern und darin die Logik eines Algorithmus abzubilden. Da sich die einzelnen Teilnehmer eines Algorithmus in der Regel gleich verhalten, genügt es oft, nur eine einzige Klasse pro Algorithmus zu entwickeln.

Um einen kurzen Einblick in die Möglichkeiten zu erhalten, die die Klasse `Agent` dem Benutzer eröffnet, werden dessen Eigenschaften vorgestellt. Jeder Agent eines Systems besitzt eine eindeutige Kennung, die im Attribut `ID` festgelegt wird. Diese Kennung kann nützlich sein, um die einzelnen Agenten voneinander zu unterscheiden. Für manche Algorithmen wird eine eindeutige Kennung sogar vorrausgesetzt. Neben einer Kennung verfügen die Agenten über ein Attribut, das ihren Zustand beschreibt. Ein möglicher Zustand wäre beispielsweise das Warten auf eine Nachricht. Ein Agent besitzt zudem eine Rolle, die er im Zusammenspiel mit anderen Agenten erfüllt. Die wichtigsten Methoden, die durch die Klasse `Agent` geerbt werden, sind Methoden zum Senden und Empfangen von Nachrichten. Die Bedeutung der Kommunikation und die des Nachrichtenflusses wird später beschrieben.

Die Topologie eines verteilten Systems ist allerdings erst mit den Verbindungen komplett. Diese werden nicht in den Agentenklassen oder in einer anderen Klasse definiert, sondern erst im Programm, wo die gesamte Umgebung konfiguriert wird. Dadurch kann der Anwender entscheiden, mit wie vielen Agenten er den Algorithmus testen will und wie diese Agenten miteinander verbunden sind, und muss sich nicht im Vorfeld darauf festlegen.

4.3 Kontrolle und Steuerung

Die Simulation in DisASTer verläuft nachrichtengesteuert. Das erklärt den hohen Stellenwert der Sende- und Empfangsmethoden der Agenten. Nachrichtengesteuert bedeutet, dass eine Aktion oder eine bestimmte Handlungsweise eines Agenten nicht durch ein Ereignis, sondern durch das Eintreffen einer Nachricht ausgelöst wird. Abbildung 4.1 zeigt, dass Nachrichten, die eine Aktion auslösen sollen, nicht von einem anderen Teilnehmer stammen müssen. Die „init"-Nachricht wird beispielsweise von keinem Agenten sondern vom System gesendet. Andernfalls kann ein Agent sich selbst benachrichtigen um bestimmte Aktionen durchzuführen. Aus der Abbildung wird ebenfalls ersichtlich, dass verschiedene Aktionen

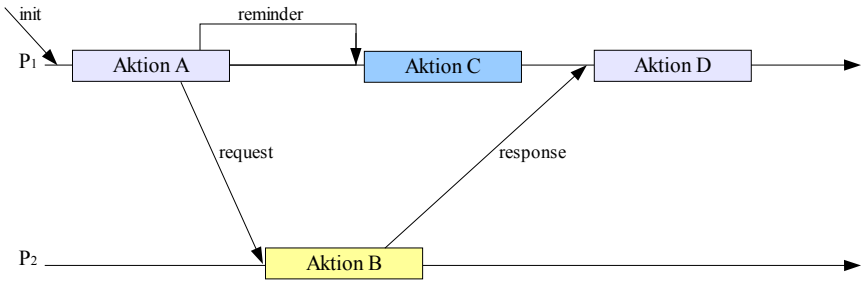


Abbildung 4.1: Nachrichtengesteuerte Simulation

durch unterschiedliche Nachrichtentypen ausgelöst werden. Das Eintreffen eines bestimmten Nachrichtentyps kann durch eine extra für sie definierte Methode abgefangen werden. Das erleichtert dem Anwender die Trennung und Auslösung der unterschiedlichsten Aktionen durch unterschiedliche Nachrichtentypen. Eine Nachricht besitzt, unabhängig von ihrem Typ, folgende Attribute:

- `sender`, der Agent, der die Nachricht verschickt hat.
- `receiver`, der Agent, der die Nachricht erhält. Ein Empfänger muss in jedem Fall angegeben werden und mit dem Sender verbunden sein, sofern sich der Sender nicht selbst eine Nachricht sendet.
- `sendTime`, beschreibt den Zeitpunkt, an dem die Nachricht gesendet wurde.
- `receptionTime`, legt zunächst fest, wann eine Nachricht ausgeliefert werden soll. Der dafür angegebene Zeitpunkt ist nicht verbindlich. Sobald die Nachricht bei ihrem Empfänger angekommen ist, erhält dieses Attribut den Zeitpunkt der tatsächlichen Auslieferung.

- `payload`, in dem sich die Informationen befinden, die ein Agent einem anderen mitteilen oder übergeben möchte.

Würden die Nachrichten sofort von ihrem Sender zum Empfänger geleitet, gäbe es für den Benutzer keine Möglichkeit, in den Ablauf der Simulation einzugreifen. Aus diesem Grund wird die MessageQueue, eine Warteschlange, in die die gesendeten Nachrichten eingereiht werden, eingeführt. Mit Hilfe dieser MessageQueue wird eine Steuerung der Kommunikation möglich. Die Arbeitsweise der MessageQueue wird in Abbildung 4.2 dargestellt.

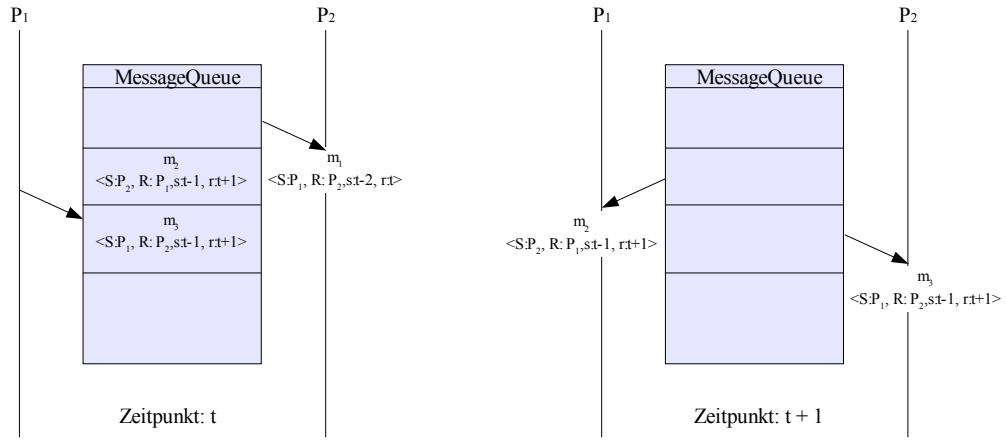


Abbildung 4.2: MessageQueue

Zunächst wird auf die automatische Steuerung der MessageQueue eingegangen. Diese erlaubt dem Benutzer, den erstellten Algorithmus wie einen Film ablaufen zu lassen. Dabei stehen ihm Steuerbefehle wie Start, Stopp, Pause oder Einzelschritt, sowie der Zugriff auf den bisherigen Verlauf der Simulation, um diesen zurück zu spulen, zur Verfügung. Abgesehen von Nachrichten, denen eine Wunschauslieferungszeit übergeben wurde, wird die MessageQueue nach dem FIFO-Prinzip ausgeleert. Das bedeutet, dass die Nachrichten, die zuerst in die MessageQueue eingegangen sind, auch zuerst ausgeliefert werden. Dabei kann der Benutzer zwischen dem synchronen und dem asynchronen automatischen Modus wählen:

1. Im synchronen Modus werden die Nachrichten quasi in Echtzeit ausgeliefert, d.h., dass alle Nachrichten, deren Ankunftzeiten mit der aktuellen Zeit übereinstimmen, auf einen Schlag aus der MessageQueue entfernt und an ihre Empfänger ausgeliefert werden.

2. Der asynchrone Modus unterteilt einen Zeitpunkt, in dem mehrere Nachrichten ausgeliefert werden sollen, in Zeitschritte. Dabei wird in jedem Zeitschritt immer nur eine Nachricht ausgeliefert. Die Ankunftzeit bleibt dabei die gleiche, um die Konsistenz zwischen dem synchronen und dem asynchronen Modus zu gewährleisten. Dieser Modus dient vor allem der Übersichtlichkeit, da in vielen Fällen die Vorgehensweise eines Algorithmus besser nachvollzogen werden kann, wenn er Nachricht für Nachricht abläuft.

Die automatische Steuerung verhindert allerdings die Willkür, die in verteilten Systemen auftritt, in denen es zu Verzögerungen oder Nachrichtenverlusten kommen kann. Um diese Störfaktoren auch in DisASTer einbringen zu können, wird eine manuelle Steuerung angeboten, die dies ermöglicht. Um die manuelle Steuerung zu gewährleisten, wird die MessageQueue aufgeteilt. Für jeden Agenten existiert eine separate MessageQueue, in der sich alle Nachrichten befinden, die an diesen Agenten adressiert sind. Um auch hier die Konsistenz zu bewahren, werden diese AgentMessageQueues von der MessageQueue, in der sich alle gesendeten Nachrichten befinden, verwaltet. Abbildung 4.3 zeigt die Aufteilung der Nachrichten in der MessageQueue in die einzelnen AgentMessageQueues.

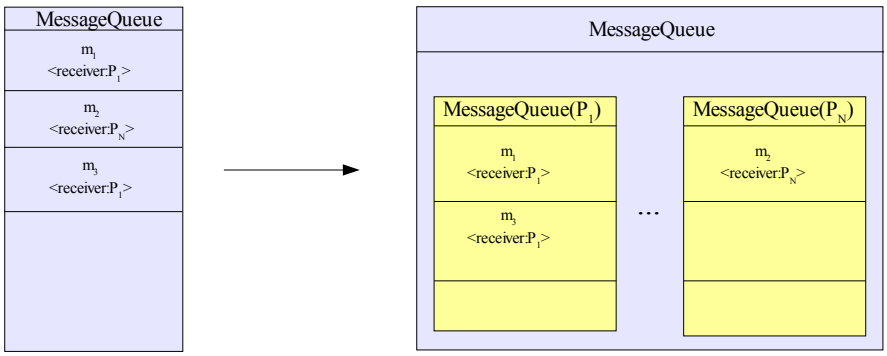


Abbildung 4.3: AgentMessageQueues

Bei der manuellen Steuerung hat der Anwender nun den vollen Zugriff auf alle Nachrichten innerhalb dieser separaten Listen, die durch die Aufteilung unabhängig von einander bearbeitet werden können. Dort kann er die Reihenfolge ändern, die Auslieferung von Nachrichten, die eigentlich zu einem späteren Zeitpunkt erfolgen sollte, kann vorgezogen werden oder es können Nachrichten aus den Listen entfernt werden. Über diese Änderungen können die oben angesprochen Szenarien, wie Nachrichtenverlust oder -verzögerung, simuliert werden, um

zu testen, wie das System auf diese Probleme reagiert.

4.4 Visualisierung

Um die Visualisierung muss sich der Anwender in der Regel nicht kümmern. Es werden bereits zwei unterschiedliche Darstellungsformen (Ansichten) angeboten:

1. SequenceView: zeigt den Algorithmus in einem Sequenzdiagramm an.
2. TopologyView: stellt die Topologie des Systems grafisch, in Form eines Graphen, dar.

Die Ansichten verwalten eine Reihe von Renderern, die zur Darstellung unterschiedlicher Elemente der Umgebung (z.B. Agenten, Verbindungen oder Nachrichten) verantwortlich sind. Abbildung 4.4 zeigt die Zusammenarbeit der Topologieansicht und ihrer Renderer.

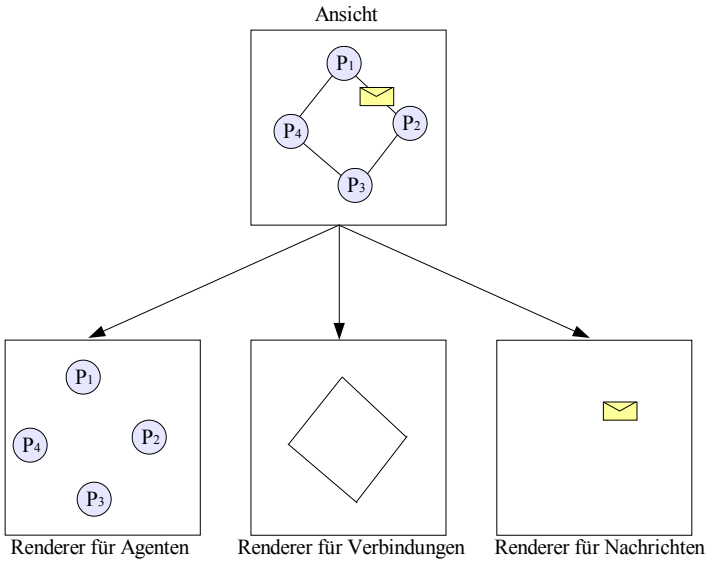


Abbildung 4.4: Ansicht und Renderer

Die bereits bestehenden Ansichten können vom Benutzer angepasst oder erweitert werden. Sollten die angebotenen Ansichten nicht ausreichen, besteht zusätzlich die Möglichkeit, eigene Ansichten zu implementieren.

5 Benutzung des Programmes

5.1 Inhalt und Installation

5.1.1 Inhalt

DisASTer wird in zwei Versionen ausgeliefert. Zum einen die Quellcode-Variante (src), in der alle wichtigen Dateien enthalten sind um das Programm neu zu erzeugen. Alternativ dazu kann die bereits fertig entfaltete Version (bin) verwendet werden. Die Versionen finden sich in zwei unterschiedlichen Zip-Archiven:

1. `disaster-src.zip` : beinhaltet die Quellcode-Variante und hat eine Größe von ca. 1.2 MB.
2. `disaster-bin.zip` : beinhaltet alle für den Programmablauf benötigten Dateien und besitzt eine Größe von ca. 8 MB.

Die Quellcode-Version besitzt folgende Dateien und Verzeichnisse:

- `doc` : in diesem Verzeichnis befindet sich das Benutzerhandbuch des Programms im HTML-Format.
- `lib` : dieses Verzeichnis beinhaltet zur Zeit nur die zusätzliche Javahelp-Bibliothek jhall.jar.
- `saved` : dieser Ordner enthält bereits gespeicherte Beispielealgorithmen.
- `src` : ist das Verzeichnis, indem die Java-Quelldateien des Programms zu finden sind.
- `DisASTer.bat` : diese Stapelverarbeitungsdatei und
- `DisASTer.sh` : können später zum Starten der Anwendung verwendet werden.
- `license.txt` : beinhaltet die Lizenz (GNU General Public License), unter der dieses Programm erhältlich ist.
- `build.xml` : ist das Ant-Skript zum Kompilieren der Quelldateien.
- `MANIFEST.MF` : wird zur Erzeugung des Jar-Archivs verwendet, in dem sich die Anwendung später befindet.
- `Readme.txt` : beinhaltet diese Informationen.

Die fertige Version des Programms enthält ebenfalls diese Struktur und noch einige zusätzliche Dateien und Ordner:

- `bin` : in diesem Verzeichnis finden sich die kompilierten Javaklassen des Programms.
- `lib` : zusätzlich zur Javahelp-Bibliothek finden sich hier zwei weitere Archive:
  - `disaster.jar` : in dem sich die Anwendung befindet.
  - `disaster-dev.jar` : in dem sich nur die zur Entwicklung nötigsten Klassen befinden.

  Neben diesen Archiven müssen alle neuen Klassen in dieses Verzeichnis gelegt werden.
- `doc` : neben dem Benutzerhandbuch findet sich in diesem Ordner nun auch die API-Dokumentation aller Klassen.

5.1.2 Systemvoraussetzungen

Um DisASTer zu verwenden, wird eine *Java Runtime Environment* bzw. ein *Java 2 Software Developement Kit* der Version 1.4 oder höher vorausgesetzt. Der Rechner, auf dem das Programm eingesetzt wird, sollte folglich die Systemvoraussetzungen besitzen, die auch schon für die Laufzeitumgebung vorgeschlagen werden; das bedeutet ein Prozessor mit mindestens 166MHz und einen Arbeitsspeicher von 48 MB. Um die Quellcode-Version zu installieren, wird Ant von Apache[2] in der Version 1.5 oder höher vorausgesetzt. Die vollständige Installation von DisASTer benötigt mindestens 16 MB freien Speicher auf der Festplatte.

5.1.3 Installation und Start

Bei beiden angebotenen Versionen ist die Installation einfach und in nur wenigen Schritten abgeschlossen.

1. Die bereits vollständige Version muss nur in ein beliebig wählbares Verzeichnis entpackt werden.
2. Um aus der Quellcode-Version das vollständige Programm zu erzeugen, wird Ant von Apache eingesetzt. Sollte sich der ant-Befehl im Pfad befinden, genügt es in das Verzeichnis zu wechseln, in dem die Version entpackt

[2] http://ant.apache.org

wurde und den Befehl ant auszuführen. Andernfalls muss beim Aufruf des ant-Befehls das enthaltene Antskript, `build.xml`, übergeben werden.

Die Anwendung kann nun über die enthaltenen Shellskripte (`DisASTer.bat`, `DisASTer.sh`) gestartet werden. Alternativ hierzu genügt ein Doppelklick auf die Datei `disaster.jar` im lib-Verzeichnis, falls das verwendete Betriebssystem den `java -jar` bzw `javaw`-Befehl mit diesem Dateityp verknüpt hat.

5.2 Die Bibliothek

In den folgenden Abschnitten wird die Klassenbibliothek vorgestellt, mit deren Hilfe ein Entwickler schnell in der Lage sein soll, eigene verteilte Algorithmen zu implementieren. In Abschnitt 5.2.1 wird die Klasse `Agent` vorgestellt. Dort wird beschrieben, welche Methoden und Mechanismen angeboten werden, die das Verhalten eines Teilnehmers eines verteilten Algorithmus bestimmen. In Abschnitt 5.2.2 wird gezeigt, was es mit den Nachrichten auf sich hat und wie diese verwendet oder erweitert werden können. Anschließend wird in Abschnitt 5.2.3 beschrieben, welche Möglichkeiten es gibt, die Agenten miteinander zu verbinden, um die Kommunikation zu gewährleisten. Zuletzt werden die Ansichten vorgestellt und erklärt, wie diese erweitert werden können.

Um Klassen der DisASTer-Bibliothek zu erweitern, muss diese in den Klassenpfad der zu erstellenden Klassen eingebunden werden. Hier besteht zum einen die Möglichkeit, die gesamte Anwendung, die sich in dem Archiv `disaster.jar` befindet, einzubinden, oder es kann das zur Entwicklung völlig ausreichende Archiv `disaster-dev.jar` verwendet werden. Letzteres enthält lediglich die Klassen des Modells und der Ansichten der Anwendung, die auch im folgenden vorgestellt werden. Um die neu erstellten Klassen, seien es eigens implementierte Agenten, Verbindungen o.a., in das Programm integrieren zu können, müssen diese im Klassenpfad der Anwendung liegen. Hierzu wird der `lib`-Ordner verwendet, der einen Teil des Klassenpfades des Programms bildet. Das bedeutet, dass alle selbst geschriebenen Klassen (`.class`-Dateien) mit samt ihrer Package-Struktur in das `lib`-Verzeichnis gelegt werden müssen.

5.2.1 Verwendung und Implementierung von Agenten

Die wichtigste Klasse bei der Entwicklung verteilter Algorithmen mit DisASTer ist die Klasse `Agent` (`disaster.model.Agent`). Ein Agent stellt in DisASTer einen Teilnehmer eines Algorithmus dar, der durch die Kommunikation mit den anderen Prozessen für den Ablauf des Algorithmus verantwortlich ist. Zunächst

beschränkt sich die Aufgabe also nur darauf, eigene Prozesse zu beschreiben, in dem die Klasse `Agent` erweitert und in der neuen Klasse das Verhalten des Prozesses festgelegt wird. Häufig ist es ausreichend, eine einzige Klasse für einen Algorithmus zu spezifizieren, da sich das Verhalten der einzelnen Prozesse innerhalb eines verteilten Algorithmus oft nicht unterscheidet.

Diese neue, aus `Agent` abgeleitete Klasse, erbt bereits viele nützliche Methoden und Attribute. Zu den wichtigsten Attributen gehören:

- `ID` : eine eindeutige Kennung, die jeder Agent besitzt. Repräsentiert wird diese Kennung durch einen einfachen `int`-Wert, der mit jeden neuen Agenten inkrementiert wird. Die Zuordnung einer Kennung erfolgt automatisch, so dass der Anwender sicher sein kann, dass jeder Agent eine andere Kennung besitzt. Zugriff auf die Kennung erfolgt über die Methode `getID()`.

- `state` : dieses Attribut beschreibt den Zustand des Agenten. Dabei handelt es sich erneut um einen `int`-Wert, dem der Benutzer vordefinierte Konstanten oder eigene Werte übergeben kann. Bereits definierte Zustände sind:

  - `STATE_NORMAL` : beschreibt den normalen Zustand eines Agenten.
  - `STATE_WAITING` : kann dazu verwendet werden, einen Agenten als wartend (z.B. auf eine Nachricht) zu kennzeichnen.
  - `STATE_WORKING` : sagt aus, dass der Agent zur Zeit beschäftigt ist.

  Die vordefinierten Zustände können auch für andere Zwecke verwendet werden. Hierbei handelt es sich nur um Vorschläge, die einen Zustandswechsel kennzeichnen können. Wichtig dabei ist, dass die Visualisierungen auf diese vordefinierten Zustände eingehen und diese grafisch darstellen können. Der Zugriff auf den Zustand wird durch die Methoden `getState()` und `setState(int newState)` gewährleistet.

- `role` : beschreibt die Rolle des Agenten. Eine Rollenbeschreibung ist vor allem im Vergleich mit anderen Agenten von Vorteil. Fungiert ein Agent als zentraler Server, als Anführer oder ähnliches, kann das durch die Rollen verdeutlicht werden. Als Konstanten vordefinierte Rollen sind:

  - `ROLE_MASTER` : zeigt an, dass dieser Agent eine übergeordnete Rolle spielt.
  - `ROLE_SLAVE` : beschreibt die Rolle eines einfachen Agenten.

  Lesender und schreibender Zugriff auf die Rolle erfolgt über die Methoden `getRole()` und `setRole(int newRole)`.

Neben diesen Attributen verfügt die Klasse Agent über viele hilfreiche Methoden. Zunächst werden die Methoden vorgestellt, die die Kommunikation mit den anderen Agenten und das Senden von Nachrichten ermöglichen:

- `send(Message message)` : sendet die übergebene Nachricht. Diese Methode kann dazu verwendet werden unterschiedliche Nachrichtentypen zu versenden, die beim Empfänger auch unterschiedlich Aktionen auslösen können. Zum Empfang dieser Nachrichten wird in diesem Abschnitt noch einiges erklärt. Bereits existierende Nachrichtentypen werden im nächsten Abschnitt vorgestellt.

- `send(Object payload, Agent receiver)` : diese und die weiteren Methoden sollen dem Anwender ein wenig die Arbeit mit dem Standardnachrichtentyp vereinfachen, so dass er nicht jedesmal ein neues Nachrichtenobjekt erzeugen muss. Über diese Methode wird eine Nachricht mit dem übergebenen Datenpaket an den angegebenen Empfänger gesendet.

- `send(Object payload, Agent receiver, int duration)` : diese Methode sendet ebenfalls eine Standardnachricht an den übergebenen Empfänger, allerdings wird diese Nachricht frühestens nach der angegebenen Zeitverzögerung ausgeliefert.

- `send(Object payload, Connection connection)` : mit dieser Methode wird ein Agent nicht direkt, sondern über die Verbindung zu ihm angeschrieben.

- `send(Object payload, Connection connection, int duration)` : diese Methode funktioniert wie die vorherige, mit dem Unterschied, dass hier wieder eine Zeitverzögerung stattfindet.

Die bisherigen Methoden werden hauptsächlich dazu verwendet, eine Nachricht an einen Empfänger zu senden. Daneben gibt es Methoden, die eine Nachricht gleichzeitig an alle, mit dem Agenten verbundenen, Agenten senden. Alternativ dazu kann bei den bisher vorgestellten Methoden die Broadcast-Adresse (`Agent.BROADCAST`) angegeben werden. Das Verhalten entspricht den hier vorgestellten:

- `sendToAll(Message message)` : mit dieser Methode wird eine Art Broadcast bzw. Multicast simuliert. Alle über bestehende Verbindungen mit dem aktuellen Agenten erreichbaren Teilnehmer erhalten eine Kopie der übergebenen Nachricht. Über diese Methode besteht die Möglichkeit, unterschiedliche Nachrichtentypen zu versenden.

- `sendToAll(Object payload)` : über diese Methode kann ein Agent ein bestimmtes Paket an alle seine erreichbaren Nachbarn schicken. Es wird der Standardnachrichtentyp verwendet.

- `sendToAll(Object payload, int duration)` : bei dieser Methode wird wieder eine Zeitverzögerung bestimmt.

Neben der Kommunikation mit anderen Agenten besitzt ein Agent die Möglichkeit sich selbst Nachrichten zu schicken, um gewisse Aktionen aus eigener Hand zu starten. Für diesen Zweck wurde ein spezieller Nachrichtentyp, der Timeout ((`disaster.model.TimeoutMessage`)) bereitgestellt. Die folgenden Methoden führen zum Senden eines Timeouts:

- `setTimeout(int duration)` : mit dieser Methode sendet der Agent eine Timeoutnachricht an sich selbst, die nach der angegebenen Zeit ankommt. Dadurch kann eine geplante Aktion zu einem späteren Zeitpunkt ausgeführt und der entsprechende Agent schlafen gelegt werden. Dieser Mechanismus sollte als Ersatz für `Thread.sleep()` verwendet werden, da sich `Thread.sleep()` auf die Systemzeit und nicht auf die Simulationszeit bezieht. Zu beachten wäre hier allerdings, dass der Agent zwischen dem Senden und dem Empfangen eines Timeouts weitere Aktionen durchführen kann.

- `setTimeout(Object payload, int duration)` : mit dieser Methode bekommt der Agent ebenfalls eine Auszeit, kann sich aber selbst noch ein bestimmtes Objekt zusenden, was er dann nach der angegebenen Zeit erhält. Im Gegensatz zu den vorangegangenen sende-Methoden muss zum Senden eines Timeouts kein Empfänger festgelegt werden, da die Nachricht an den Sender zurückkommt. Es ist aber auch möglich, eine Timeoutnachricht an einen anderen Agenten zu senden. Dies geschieht dann über die einfache sende-Methode (`send(Message)` bzw. `send(TimeoutMessage)`).

Nachdem bekannt ist, wie andere Agenten angesprochen werden, wird auf die Methoden eingegangen, die beim Eintreffen dieser Nachrichten aufgerufen werden. Für diesen Zweck bietet DisASTer einen Pseudopolymorphismus an. Beim Eintreffen einer Nachricht wird die Methode `onReception(Message)` aufgerufen. Diese Methode kann für alle Nachrichtentypen „überschrieben“ werden. Um z.B. das Eintreffen einer Anfrage (`Request`) gesondert zu behandeln, wird die Methode `onReception(Request)` definiert, in der beschrieben wird, was der Agent nach Erhalt einer Anfrage machen soll. Existiert für die ankommende Nachricht keine entsprechende `onReception()`-Methode, wird die Standardmethode `onReception(Message)` ausgeführt. Durch diesen Mechanismus können verschiedene Aktionen mit Hilfe unterschiedlicher Nachrichtentypen ausgelöst werden.

Der Anwender ist an dieser Stelle nicht nur auf die bereits existierenden Nachrichtentypen angewiesen, sondern kann seine eigenen Typen definieren, was im nächsten Abschnitt (Verwendung und Implementierung von Nachrichten) beschrieben wird.

Die Differenzierung der Nachrichtentypen kommt bereits zu Beginn eines verteilten Algorithmus zum Tragen. Um einen Algorithmus in Gang zu setzen, werden Initialisierungsnachrichten (`InitMessages`) verwendet, die an einen oder mehrere Agenten adressiert sind. Beim Eintreffen einer solchen Nachricht wird die `init()`-Methode des Empfängers aufgerufen, in der er weitere Nachrichten an seine Nachbarn senden kann, um damit eine Kettenreaktion auszulösen. Dadurch sind diese ersten Initialisierungen die Voraussetzung für den Ablauf eines Algorithmus. Anders als bei Applets oder Servlets können Agenten auch mehrmals initialisiert werden. Welcher Agent zu welchem Zeitpunkt initialisiert werden soll, wird bei der Einrichtung der Umgebung festgelegt.

Um seinen Agenten einmal eine kleine Auszeit zu gönnen oder eine bestimmte Aktion verzögert ausführen zu lassen, kommt ein weiterer Nachrichtentyp zum Einsatz. Über die `setTimeout()`-Methoden sendet ein Agent eine Nachricht vom Typ `disaster.model.TimeoutMessage` an sich selbst. Nach einer angegebenen Zeit wird die Methode `timeout(TimeoutMessage)` aufgerufen, über die diese Nachricht verarbeitet werden kann.

Eine Broadcast-Nachricht, ob über `sendToAll()` oder `send(Agent.BROADCAST)` gesendet, erhalten grundsätzlich nur alle erreichbaren Nachbarn des sendenden Agenten. Ein erreichbarer Nachbar hat die Eigenschaft mit diesem Agenten verbunden zu sein. Diese Verbindungen werden in einer Hashtabelle der Klasse `HashMap` verwaltet und können über die Hashtabelle selbst oder über die Methode `getOthers()` abgefragt werden. Eine Hashtabelle wird mit Schlüssel-Wert-Paaren gefüllt, wobei eine Verbindung den Wert darstellt und der Agent, zu dem diese Verbindung führt, den Schlüssel. Dadurch wird garantiert, dass es nur eine Verbindung zu einem Agenten gibt.

Um weitere Informationen anzuzeigen oder auch individuelle Einstellungen vorzunehmen, gibt es die Möglichkeit, eigene Swing-Komponenten zu jedem Agenten zu definieren. Die entsprechende grafische Oberfläche wird in einer Klasse, die von `AgentComponent` abgeleitet ist, beschrieben. Eine solche Komponente leitet sich aus `JComponent` ab und besitzt eine Referenz auf einen Agenten. Von diesem können Informationen abgegriffen werden, die über die grafische Oberfläche angezeigt oder sogar verändert werden können. Für den Aufbau dieser Oberfläche stehen dem Benutzer alle Klassen der Swing-Bibliothek zur Verfügung. Allerdings muss die Serialisierbarkeit der Oberfläche gewährleistet werden.

Damit wären alle Schritte erklärt, die zur Entwicklung von Agenten verteilter Algorithmen notwendig sind. Um Programme zu testen und zu debuggen oder einfach nur ein paar Informationen zu erhalten, greifen viele Entwickler auf

Textausgaben zurück. Die eben erstellten Agenten bilden hier keine Ausnahme. Wer es gewöhnt ist, kann beim altbewärten `System.out.println()` bleiben. Die Standardausgabe (`System.out`) und Standardfehlerausgabe (`System.err`) werden für diesen Zweck in das Programm umgeleitet. Ein eleganterer Weg wäre jedoch der Einsatz der `println()`-Methode eines Agentenobjekts. Die hier erfolgten Eingaben werden in eine extra Konsole geleitet, wobei jeder Eintrag eines bestimmten Agenten farblich hervorgehoben wird. Eine genauere Beschreibung dieser Konsole und der weiteren Vorteile gegenüber der Standardausgabe findet sich in Abschnitt 5.3.2.

Allgemein ist bei der Entwicklung eines Agenten darauf zu achten, dass zusätzliche Klassen serialisierbar sind. Gerade beim Einsatz von Swing ist dies nicht immer ganz einfach. Um jedoch den vollen Funktionsumfang von DisASTer nutzen zu können ist die Serialisierbarkeit der verwendeten Klassen Pflicht.

5.2.2 Verwendung und Implementierung von Nachrichten

Bei der Vorstellung der Agenten und ihrer Arbeitsweise wurde bereits ersichtlich, dass Nachrichten und unterschiedliche Nachrichtentypen ein weiterer wesentlicher Bestandteil der Entwicklung verteilter Algorithmen mit DisASTer sind. Das Eintreffen einer Nachricht ruft in der Regel eine bestimmte Aktion beim Empfänger auf. Um unterschiedliche Aktionen auszulösen, können unterschiedliche Nachrichtentypen verwendet werden. Im Package `disaster.model` befinden sich bereits einige vordefinierte Typen:

- `Message` : Standardnachricht, von der alle anderen Nachrichtentypen abgeleitet werden. Als Reaktion auf das Eintreffen einer `Message` wird die Methode `onReception(Message)` aufgerufen.
- `Request` : ein `Request` kann dazu verwendet werden Anfragen an andere Agenten zu stellen. Als Reaktion auf das Eintreffen einer `Request` wird die Methode `onReception(Request)` aufgerufen.
- `Response` : ein `Response` kann dazu verwendet werden Antworten auf eine andere Nachricht zu geben. Als Reaktion auf das Eintreffen einer `Response` wird die Methode `onReception(Response)` aufgerufen.
- `InitMessage` : Spezialfall. Dieser Nachrichtentyp initialisiert einen Agenten. Beim Empfang einer `InitMessage` wird die Methode `init()` aufgerufen. Eine Nachricht dieses Typs kann wie die anderen auch an einen anderen Agenten gesendet werden.
- `TimeoutMessage` : ein weiterer Spezialfall. Nach Ablauf des Timeouts wird die Methode `timeout()` aufgerufen. Auch dieser Nachrichtentyp kann an

andere Agenten gesendet werden.

Der Unterschied zwischen den einzelnen Nachrichtentypen besteht weitestgehend nur darin, dass sie unterschiedliche Klassen repräsentieren und durch den Pseudopolymorphismus entsprechende `onReception()`-Methoden aufrufen, um bei dem Empfänger unterschiedliche Aktionen auszulösen. Geht eine Nachricht ein, für dessen Typ keine explizite Empfangsmethode deklariert wurde und dies ebenfalls für die Nachrichtentypen der Fall ist, aus denen dieser Typ abgeleitet ist, wird die Nachricht von der Standardmethode (`onReception(Message)`) angenommen.

Bei der Verwendung eigener Nachrichtentypen sollte auf die Verwendung eigener Attribute verzichtet werden. Diese könnten z.B. dazu verwendet werden, zusätzliche Informationen in die Nachricht zu integrieren. Die zu sendende Nachricht wird allerdings vor dem Senden kopiert, wobei alle bekannten Attribute, wie der Sender, der Empfänger, die Sende- und Empfangszeit, sowie das übergebene Datenpaket übernommen werden. Selbstdefinierte Attribute werden bei diesem Verfahren übergangen, da sie dem System nicht bekannt sind. Diese Einschränkung wurde gewählt, um dem Entwickler zu ersparen, die `clone()`-Methode zu überschreiben. Zusätzlichen Informationen sollten statt in den neuen Nachrichtentyp in das versendete Paket gesteckt werden. Für dieses Paket kann dann eine neue Klasse definiert werden, die alle benötigten Zusatzinformationen beinhaltet. Dieses Paket wird dann auch in die jeweilige Kopie der Nachricht gepackt. Allein die Serialisierbarkeit muss gewährleistet sein. Gegenüber dem Überschreiben der `clone()`-Methode ist dieser Schritt mit geringerem Aufwand für den Anwender verbunden.

5.2.3 Verwendung und Implementierung von Topologiemanagern und Verbindungen

Nachdem die Agenten vorgestellt wurden und der Umgang mit Nachrichten erklärt wurde, soll nun darauf eingegangen werden, welche Schritte notwendig sind, um die Kommunikation zwischen den einzelnen Agenten zu gewährleisten. Denn eine Kommunikation zwischen zwei Agenten kann nur stattfinden, wenn beide miteinander verbunden sind. Solch eine Verbindung stellt die Klasse `Connection` (`disaster.model.Connection`) dar. In Abschnitt 5.3.3 wird erklärt, wie solche Verbindungen manuell erzeugt werden können. Es gibt allerdings auch die Möglichkeit, Verbindungen nach festen Regeln automatisch erzeugen zu lassen. Diese Arbeit übernehmen die Topologiemanager (`disaster.model.TopologyManager`). Sie legen fest, welche Agenten miteinander verbunden werden und welche nicht. Zusätzlich kann ein Topologiemanager die Positionen dieser Agenten festlegen,

wobei diese bisher ausschließlich für die Visualisierung durch die Topologieansicht verwendet werden. In DisASTer sind bereits zwei solche Topologiemanager zu finden:

1. Zum einen die `CompleteTopology`, die jeden Agenten mit jedem anderen verbindet. In dieser Topologie erreicht jeder Agent alle anderen über die Broadcastadresse.

2. Zum anderen der `TokenRing`, der die Agenten ringförmig anordnet. Die Kommunikation in einem Token Ring verläuft in der Regel nur einseitig. Um dieses Verhalten zu erreichen, verwendet der `TokenRing` eine eigene `Connection`-Klasse, die nur eine einseitige Kommunikation zulässt. Um eine Nachricht auf diesen Token Ring zu senden, muss die Broadcastadresse verwendet werden. Mit ihr wird versucht eine Nachricht an alle benachbarten Agenten zu senden. Da das in diesem Fall aber nur in einer Richtung zugelassen wird, erreicht die gesendete Nachricht nur den direkten Nachbarn in Ringrichtung.

Auch hier steht es dem Benutzer offen, eigene Verbindungen oder sogar eigene Topologiemanager zu verwenden. Bei den Verbindungen wird dieser Schritt aber nicht nötig sein. Da ein Objekt der Klasse `Connection` im Grunde nur verwendet wird, um eine Verbindung zwischen zwei Agenten herzustellen, ist sie für ein weiteres Eingreifen in den Algorithmus nicht mehr von Interesse. Die einzige Aufgabe einer Verbindung besteht darin, festzulegen, welche Agenten sich untereinander Nachrichten senden können. Eigene Topologiemanager haben schon eher einen Einfluss auf den aktuellen Algorithmus. Sie bekommen eine Referenz auf die von der Umgebung verwendeteten Topologie (`disaster.model.Topology`), die alle Agenten und deren Verbindungen beinhaltet. Über diese Topologie kann der Manager in den Methoden `doConnections()` und `doPositions()` die Verbindungen und die Positionen der einzelnen Agenten bestimmen. Diese Methoden können vom Entwickler eines neuen Topologiemanagers überschrieben werden. Das Einbinden eigener Verbindungen oder Topologiemanager in das Programm wird in Abschnitt 5.3.3 beschrieben.

5.2.4 Verwendung und Implementierung von Ansichten

Die Hauptaufgabe von DisASTer ist die grafische Darstellung von verteilten Algorithmen. Diese Aufgabe wird von den Ansichten übernommen. Eine Ansicht (`disaster.view.View`) ist eine Swing-Komponente (`JComponent`), die auf mehrere Objekte Zugriff hat, welche die unterschiedlichsten Elemente einer Simulation auf diese Komponente zeichnen. Ein solches Zeichenobjekt wird als Renderer

(`disaster.view.Renderer`) bezeichnet. Elemente, die von einem solchen Renderer gezeichnet werden können sind z.B. Agenten, Nachrichten oder Verbindungen. Die Arbeitsweise der Ansichten und der Renderer wurde bereits in Abbildung 4.4 auf Seite 39 dargestellt und im entsprechenden Abschnitt erläutert. DisASTer stellt bereits zwei separate Ansichten zur Verfügung, die einen Algorithmus aus zwei unterschiedlichen Perspektiven darstellen. Bei diesen Ansichten handelt es sich zum einen um die Ablaufansicht (`disaster.view.SequenceView`), die den zeitlichen Ablauf des Algorithmus in Form eines Sequenzdiagramms darstellt. Daneben ist die Topologieansicht (`disaster.view.TopologyView`) zu finden, deren Schwerpunkt auf der Darstellung der Beziehungen, der einzelnen Agenten untereinander, sowie auf der animierten Visualisierung der Nachrichten liegt. Für diese Ansichten gibt es bereits viele Einstellungsmöglichkeiten, die Darstellung nach eigenen Wünschen anzupassen. Durch die Verwendung unterschiedlicher Farben und Formen werden z.B. die Zustände und Rollen der Agenten in der Visualisierung umgesetzt (siehe dazu auch Abschnitt 5.3.3).

Aber auch hier ist der Anwender nicht auf die bestehende Auswahl angewiesen. Wenn andere Teile einer Simulation, wie z.B. die Messagequeue, in die Visualisierung mit integriert oder einfach nur bestehende Teile individuell angepasst werden sollen, steht es frei, eigene Renderer zu implementieren. Ein Renderer besteht im wesentlichen aus drei Komponenten:

1. seiner Position und seinen Abmessungen, die mit `x`, `y`, `width`, `height` und den entsprechenden Zugriffsmethoden definiert werden,

2. einer `draw()`-Methode, in der die Zeichenoperationen festgelegt werden, um das entsprechende Element grafisch darzustellen und

3. einer Swingkomponente, über die der Renderer modifiziert werden kann.

Um die Position und die Abmessungen zu bestimmen wird ein Rechteck aufgezogen, das seine linke obere Ecke an dem von `x` und `y` beschriebenen Punkt hat und eine Breite des Wertes `width` und eine Höhe des Wertes `height` besitzt. Eine Ansicht, die diesen Renderer verwendet, hält den Platz, den dieses Rechteck beschreibt, frei und passt seine Größe gegebenenfalls an. Es ist auch möglich, ausserhalb der angegebenen Grenzen zu zeichnen. Allerdings ist dann nicht sichergestellt, dass dieser Teil auch wirklich dargestellt wird, da der Platz nicht von der Ansicht reserviert wurde. Die Grenzen mehrerer Renderer können sich durchaus auch überdecken. In diesem Fall spielt die Reihenfolge, in der die Renderer aufgerufen werden eine Rolle, da sie sich gegenseitig übermalen können.

Die eigentliche Arbeit des Renderers verrichtet die Methode

```
draw(Graphics g, double progress)
```

. Gezeichnet wird auf das übergebene `Graphics`-Objekt, das selbst schon viele Methoden bereitstellt, um einfache Formen zu zeichnen. Eine umfassende Hilfe über den Umgang mit diesem Objekt bieten die API-Dokumentation[3], das Sun-Tutorial[4] oder ein gutes Javabuch[5]. Ein wenig komplizierter ist hierbei der übergebene `double`-Wert. Falls eine Simulation animiert abläuft, zeigt dieser Wert den Fortschritt der Animation in einem Zeitschritt an. Die `draw()`-Methode wird pro Zeitschritt mehrfach aufgerufen, wobei sich der Wert von `progress` gleichmäßig von 0.0 bis 1.0 erhöht. Wird keine Animation verwendet, hat `progress` den Wert 1.0. Zur Veranschaulichung wird ein kleines Beispiel gezeigt, dass eine Linie, während eines Zeitschritts, in der Mitte des Renderers zeichnet, die mit jedem Animationsschritt länger wird und nach Ablauf der Animation am rechten Rand des Renderers angekommen ist.

```
public void draw(Graphics g, double progress) {
  int yPosition = y + height/2;
  int linewidth = x + (int) (width*progress);
  g.drawLine(x, yPosition, linewidth, yPosition);
}
```

Bei einer Animation wird diese Methode sehr oft hintereinander ausgeführt, wobei sich `progress` ständig erhöht. Je höher der Wert von `progress` wird, desto länger wird die Linie. Auf diesem Weg können viele Vorgänge durch Bewegungen veranschaulicht werden.

Um bestimmte Einstellungen an einem Renderer vorzunehmen wird eine Swing-Komponente verwendet. Die Methode `showRendererPreferences()` der Klasse `Renderer` soll diese Komponente zurückliefern. Die Einstellungen, die hier vorgenommen werden können, unterscheiden sich von Renderer zu Renderer. Eine mögliche Einstellung wäre z.B. eine Farbwahl für den Zustand eines Agenten oder eine Linienart für Verbindungen. An dieser Stelle steht es aber frei welche Einstellungen letztendlich zur Verfügung gestellt werden. Das Bereitstellen einer solchen Komponente ist zudem nicht zwingend notwendig.

Sollten die vorhanden Ansichten und die Möglichkeit, eigene Renderer in diese Ansichten zu integrieren, noch nicht ausreichen, können ohne weiteres eigene Ansichten entwickelt und zu den bereits bestehenden hinzugefügt werden. Mit dieser Strategie können direkt viele eigene Renderer, die in einer neuen Ansicht liegen, über diese eingebunden werden. Ein solcher Schritt erspart ein erneutes Anpassen der bestehenden Ansichten nach jedem Programmstart. Durch das Ableiten der Klasse `disaster.view.View` stehen schon alle wichtigen Methoden und Funktionalitäten, die eine Ansicht benötigt, zur Verfügung. Damit die neue Ansicht nicht ohne Renderer in das System integriert wird, werden ihr mit Hilfe der

[3] http://java.sun.com/j2se/1.4.1/docs/api
[4] http://java.sun.com/docs/books/tutorial/2d/index.html
[5] z.B.: http://www.javabuch.de/

Methode `addRenderer(Renderer)`, innerhalb des Konstruktors, die benötigten Renderer hinzugefügt, was im folgenden Beispiel verdeutlicht wird:

```
import disaster.view.View;
public class MyView extends View {
  public MyView() {
    super("MyView");

    addRenderer(new MyAgentRenderer());
    addRenderer(new MyMessageRenderer());
  }
}
```

Zunächst werden die Renderer in der Reihenfolge gezeichnet, in der sie der Ansicht hinzugefügt wurden. Diese Reihenfolge kann später im Programm wieder geändert werden.

Die Integration eigener Ansichten oder eigener Renderer erfolgt im laufenden Programm (Abschnitt 5.3.3, Seite 64).

5.3 Die Oberfläche

Dieser Abschnitt beschäftigt sich mit der grafischen Oberfläche von DiSASTer. Diese setzt sich aus Informationsfenstern zusammen, die Aufschluss über den Zustand des Systems geben oder die Visualisierung anzeigen, aus den Oberflächenelementen, die den Ablauf eines Algorithmus ermöglichen und diversen Einstellungsdialogen, mit denen die Umgebung oder die Visualisierung angepasst werden kann.

5.3.1 Dateioperationen

Die Dateioperationen finden sich in dieser Form in nahezu allen gängigen Programmen unter dem Menüpunkt „Datei“ bzw. „File“ wieder (Abbildung 5.1).

Zu den Dateioperationen zählen:

- New : ein neues verteiltes System wird erzeugt. Bei Aufruf dieses Menüeintrags wird der Einstellungsdialog (Abschnitt 5.3.3) gezeigt, über den das System, in der ein Algorithmus agiert, konfiguriert wird.
- Open : bereits gespeicherte Algorithmen, sogenannte *DisASTer Stored Environment*s, können mit diesem Befehl geladen werden.
- Save : mit diesem Menüeintrag wird das aktuelle System als *DisASTer Stored Environment* gespeichert. Dabei werden die wichtigsten Objekte

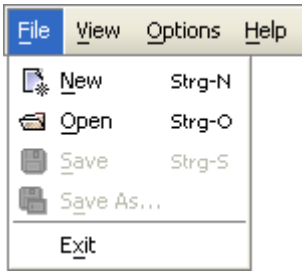


Abbildung 5.1: Dateioperationen

des Systems serialisiert. Darunter fallen die Topologie, bestehend aus den einzelnen Agenten und ihren Verbindungen, die MessageQueue, die bereits gesendeten Nachrichten und der bisherige Verlauf des Algorithmus.

- Save As... : dieser Menüeintrag speichert die momentane Umgebung in eine neue Datei.
- Exit : über diesen Menüeintrag wird das Programm beendet.

5.3.2 Informationsfenster

Hinter den Befehlen im Menüpunkt „View“ verbergen sich viele nützliche Fenster, die Aufschluss über den momentanen Zustand der Umgebung geben (siehe Abbildung 5.2).

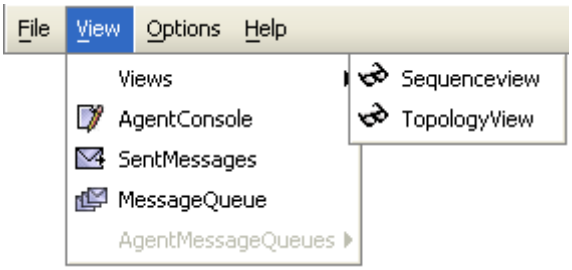


Abbildung 5.2: Informationsfenster

Es handelt sich dabei um:

- Views : in diesem Menü werden die Ansichten der Visualisierung aufgeführt. Hier befinden sich auch die vom Anwender erstellten Ansichten, nachdem diese dem System hinzugefügt wurden.

  - SequenceView : zeigt eine Ansicht, um den zeitlichen Ablauf der Simulation zu verdeutlichen.
  - TopologyView : ruft die Topologieansicht auf, die eine Draufsicht der aktuellen Umgebung liefert.

- AgentConsole : gibt die Testausgaben der einzelnen Agenten aus, die über die Methode `println()` erzeugt wurden.

- SentMessages : in diesem Fenster wird eine Liste mit den bereits ausgelieferten Nachrichten angezeigt.

- MessageQueue : in diesem Fenster werden alle Nachrichten aufgeführt, die zwar schon gesendet, aber noch nicht an den betreffenden Empfänger ausgeliefert wurden.

- AgentMessageQueues : dieses Menü gewährt Zugriff auf die Messagequeues der einzelnen Agenten, d.h. alle Nachrichten, die an einen speziellen Agenten adressiert sind, finden sich in einer separaten Liste wieder. Neben dieser Liste werden aber auch Zugriffsbefehle angezeigt, um die Nachrichtenreihenfolge zu ändern oder Nachrichten zu löschen. In diesem Fenster wird zusätzlich die agentenspezifische Swing-Komponente angezeigt, sofern diese vom Entwickler realisiert wurde.

Neben diesen Fenstern, die im Desktopbereich der Anwendung angezeigt werden, gibt es noch zwei weitere Komponenten, die Informationen über das System liefern.

- Ausgabeströme : die Standardausgabe (`System.out`) und die Standardfehlerausgabe (`System.err`) werden in das Programm umgeleitet und befinden sich unterhalb der Desktopoberfläche in zwei durch Reiter getrennten Textfeldern.

- Statusleiste : wie viele Agenten sich in einer Umgebung befinden, wie viele Nachrichten in der Messagequeue liegen, wie viele Nachrichten gesendet wurden und an welchem Zeitschritt die Simulation angekommen ist, wird in der Statusleiste angezeigt.

In den folgenden Abschnitten werden diese Informationsfenster näher erläutert.

Views Die Ansichten sind wohl die wichtigste Schnittstelle zwischen dem Anwender und DisASTer. Sie stellen den Ablauf eines Algorithmus grafisch dar. In DisASTer sind bereits zwei Ansichten integriert, die diese Aufgabe übernehmen. Es handelt sich hierbei zum einen um die Ablaufansicht (`SequenceView`, siehe Abbildung 5.3), die den zeitlichen Ablauf eines Algorithmus in Form eines Sequenzdiagramms visualisiert. Es wird gezeigt, wann eine Nachricht von wem gesendet wurde und wann diese bei welchem Empfänger angekommen ist. Zusätzlich werden Rolle und Status jedes Agenten durch verschiedene Darstellungen und Farben hervorgehoben. Ebenso kann der Nachrichtentyp durch eine bestimmte Farbgebung hervorgehoben werden.

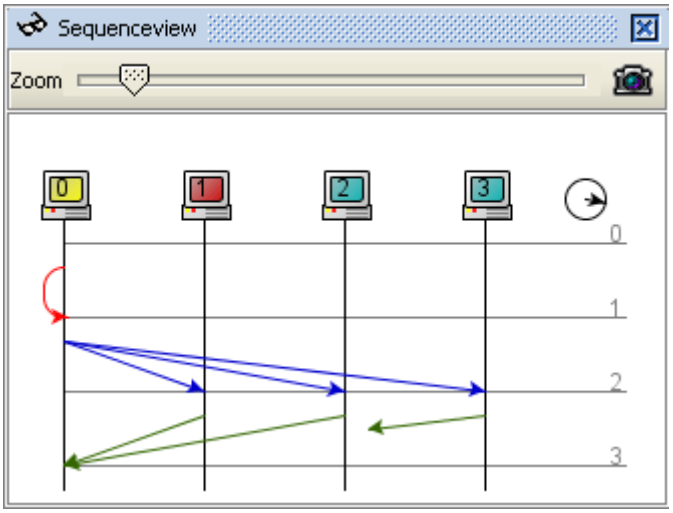


Abbildung 5.3: Ablaufansicht

Die Topologieansicht (`TopologyView`, Abbildung 5.4) ist die zweite Ansicht, die zur Verfügung steht. Sie geht auf die Verbindungen der einzelnen Agenten untereinander ein und zeigt die Nachrichten an, die gerade über die Leitungen laufen. Aus diesem Grund ist anzuraten, die Simulation animiert ablaufen zu lassen, wenn die Topologieansicht verwendet wird. Wie auch in der Ablaufansicht können sowohl Status und Rolle der Agenten als auch der Nachrichtentyp durch unterschiedliche Darstellungen hervorgehoben werden. Zusätzlich kann das Aussehen der einzelnen Verbindungen geändert werden.

Bei allen Ansichten sind noch weitere nützliche Zusatzfunktionen integriert. Mit Hilfe eines Schiebereglers kann die Ansicht vergrößert oder verkleinert werden. Die Ansicht selbst kann auch als Grafik exportiert werden (Symbol). Mögliche Formate hierbei sind die freien und plattformunabhängigen Formate

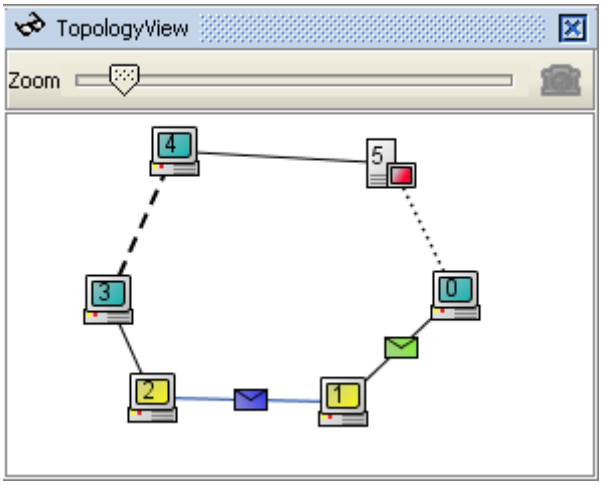


Abbildung 5.4: Topologieansicht

JPEG oder PNG. Die aktuelle Zoomeinstellung wird beim Exportieren der Ansicht berücksichtigt.

AgentConsole Zusätzlich zu der Standardausgabe (`System.out`), die sehr gerne für Testausgaben verwendet wird, bietet DisASTer eine eigene Funktionalität an. Statt der üblichen `System.out.println()`-Methode kann auch die `println()`-Methode der Klasse Agent verwendet werden. Diese Methode funktioniert ähnlich, bringt allerdings noch einige Vorteile mit sich. Die Ausgabe wird in einem separaten Fenster (Abbildung 5.5) angezeigt, wobei die Texte der einzelnen Agenten durch unterschiedliche Farben hervorgehoben werden. Neben dieser übersichtlichen Gestaltung besteht ein weiterer Vorteil darin, dass die mit `println()` erstellten Ausgaben Teil der Simulation sind und mit dem restlichen Datenmodell abgespeichert werden. Anders als bei der Standardausgabe werden so alte Ausgaben entfernt und nur die aktuellen angezeigt. Wenn nun ein bereits abgearbeiteter Zustand aus dem Verlauf wieder hergestellt wird, werden die Texte, die zeitlich hinter dem jetzt aktuellen Schritt liegen, nicht mehr angezeigt. Bei der Verwendung der Standardausgabe wäre dies nicht der Fall, sondern die Texte würden einfach an den bereits bestehenden angehängt. Somit werden durch die Agentenkonsole ebenfalls doppelte Ausgaben verhindert. Die Farbgebung der unterschiedlichen Agenten und die Abhängigkeit zum aktuellen Zeitpunkt tragen damit vornehmlich zur Übersichtlichkeit bei.

SentMessages Die Ansichten sind wichtige Informationsquellen, um zu erfahren, was während dem Ablauf des Algorithmus vor sich geht. Neben der grafischen

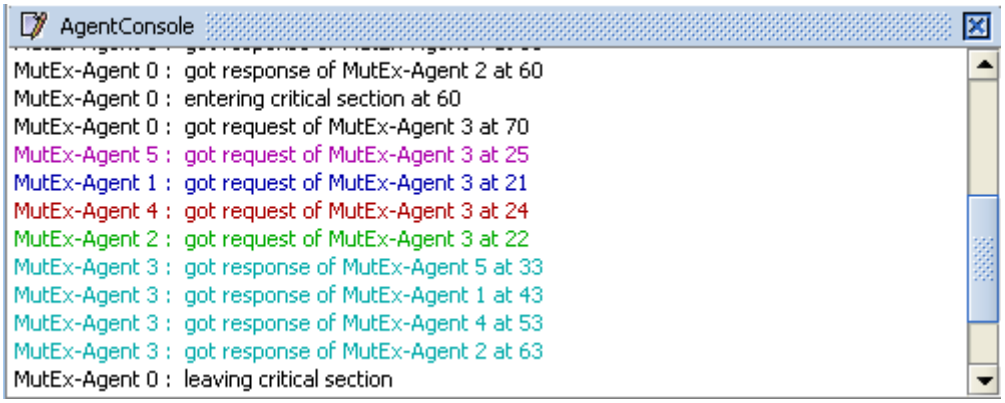
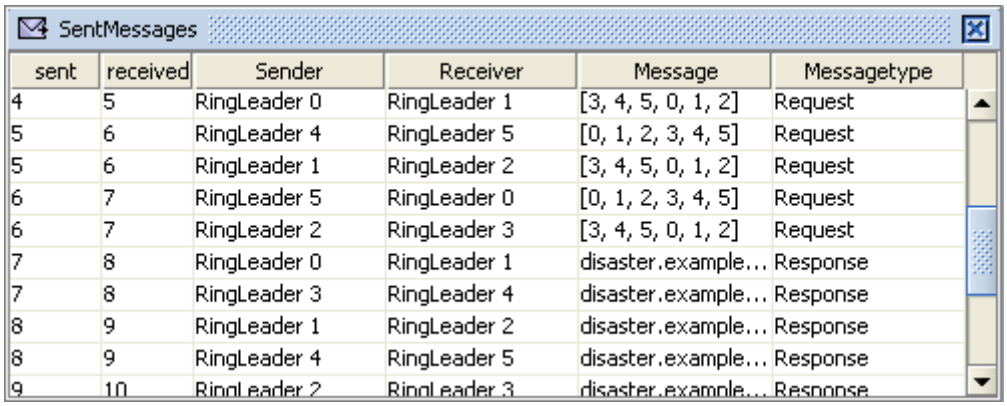

AgentConsole
MutEx-Agent 0 : got response of MutEx-Agent 2 at 60
MutEx-Agent 0 : entering critical section at 60
MutEx-Agent 0 : got request of MutEx-Agent 3 at 70
MutEx-Agent 5 : got request of MutEx-Agent 3 at 25
MutEx-Agent 1 : got request of MutEx-Agent 3 at 21
MutEx-Agent 4 : got request of MutEx-Agent 3 at 24
MutEx-Agent 2 : got request of MutEx-Agent 3 at 22
MutEx-Agent 3 : got response of MutEx-Agent 5 at 33
MutEx-Agent 3 : got response of MutEx-Agent 1 at 43
MutEx-Agent 3 : got response of MutEx-Agent 4 at 53
MutEx-Agent 3 : got response of MutEx-Agent 2 at 63
MutEx-Agent 0 : leaving critical section

Abbildung 5.5: Agentenkonsole

ist aber auch die rein textuelle Darstellung wichtig. Zusätzlich zur Agentenkonsole, in der textuelle Informationen der Agenten ausgegeben werden können, steht eine textuelle Aufführung der bereits ausgelieferten Nachrichten zur Verfügung (Abbildung 5.6).



sent	received	Sender	Receiver	Message	Messagetype
4	5	RingLeader 0	RingLeader 1	[3, 4, 5, 0, 1, 2]	Request
5	6	RingLeader 4	RingLeader 5	[0, 1, 2, 3, 4, 5]	Request
5	6	RingLeader 1	RingLeader 2	[3, 4, 5, 0, 1, 2]	Request
6	7	RingLeader 5	RingLeader 0	[0, 1, 2, 3, 4, 5]	Request
6	7	RingLeader 2	RingLeader 3	[3, 4, 5, 0, 1, 2]	Request
7	8	RingLeader 0	RingLeader 1	disaster.example...	Response
7	8	RingLeader 3	RingLeader 4	disaster.example...	Response
8	9	RingLeader 1	RingLeader 2	disaster.example...	Response
8	9	RingLeader 4	RingLeader 5	disaster.example...	Response
9	10	RingLeader 2	RingLeader 3	disaster.example...	Response

Abbildung 5.6: Gesendete Nachrichten

Die einzelnen Nachrichten werden in einer Tabelle zusammengefasst, die anzeigt, wann die Nachrichten gesendet und empfangen wurden, wer der Sender und wer der Empfänger war, welches Objekt mit dieser Nachricht verschickt wurde und welchem Typ die Nachricht zugeordnet werden konnte.

MessageQueue Nachrichten, die zwar gesendet, jedoch noch nicht ausgeliefert wurden, finden sich in der MessageQueue (Abbildung 5.7) wieder.

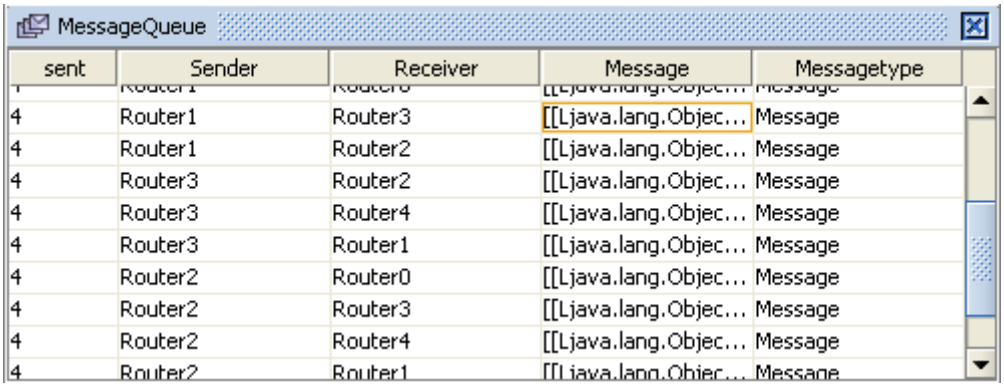


Abbildung 5.7: Messagequeue

In einer Tabelle wird angezeigt, wann eine Nachricht gesendet wurde, von wem sie stammt und an wen sie ausgeliefert werden soll. Des weiteren ist erkennbar, welches Objekt verschickt und welcher Nachrichtentyp verwendet wurde. Diese Tabelle gibt Aufschluss über den geplanten Ablauf des Algorithmus, da dort erkennbar wird, welche Aktion von welchem Agenten als nächstes ausgeführt wird.

AgentMessageQueues Für jeden Agenten existiert eine separate MessageQueue, die alle Nachrichten enthält, die an diesen Agenten adressiert sind. Dadurch kann der Benutzer sein Augenmerk auf einzelne Prozesse eines Algorithmus richten. In diesen Fenstern (Abbildung 5.8) befinden sich zusätzlich die Steuerelemente, die benötigt werden, um in den Ablauf eines Algorithmus einzugreifen. Mit Hilfe dieser Befehle kann die Nachrichtenreihenfolge ebenso geändert wie Nachrichten gelöscht werden. Im Abschnitt 5.3.4 wird genauer auf dieses Thema eingegangen.

Abgesehen von den Nachrichten, die an den speziellen Agenten gerichtet sind, wird in diesem Fenster noch zusätzlich die Agentenkomponente (`AgentComponent`) angezeigt. Bei der Agentenkomponente handelt es sich um die Swing-Komponente, die verwendet werden kann um zusätzliche Informationen oder Befehle in die grafische Benutzeroberfläche einzubinden. Der Vorteil dieser Anordnung ist, dass alle Informationen eines Agenten in einem Fenster aufgeführt werden, anstatt diese Informationen in weiteren Fenstern zu präsentieren, wodurch die Desktopoberfläche unnötig überfüllt werden würde.

Ausgabeströme Neben den Fenstern, die auf der Desktopoberfläche der Anwendung angezeigt werden, befinden sich am unteren Rand zwei durch Reiter

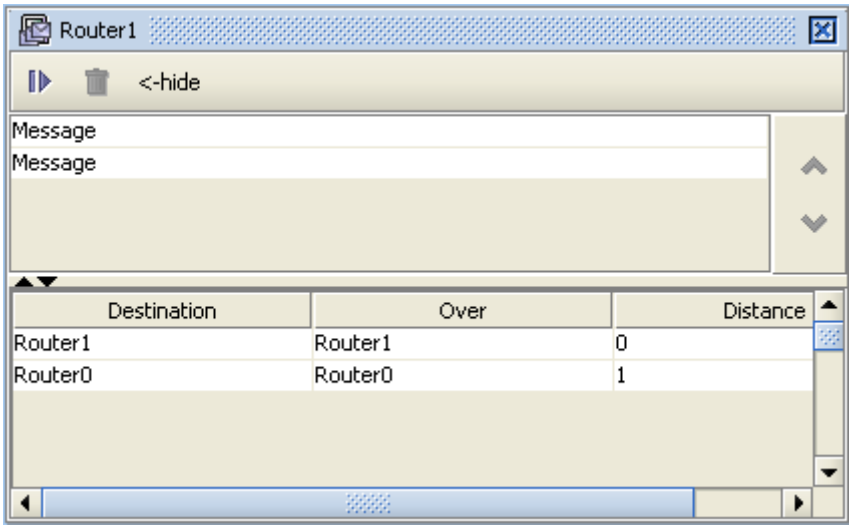


Abbildung 5.8: Agentenmessagequeue

getrennte Textfelder, in welche die Standardausgabe (`System.out`) und die Standardfehlerausgabe (`System.err`) gelenkt werden (Abbildung 5.9). Dadurch muss der Anwender nicht auf die gewohnten Ausgaben oder Stacktraces verzichten, die zum Testen und zur Fehlerbehebung sehr hilfreich sind.

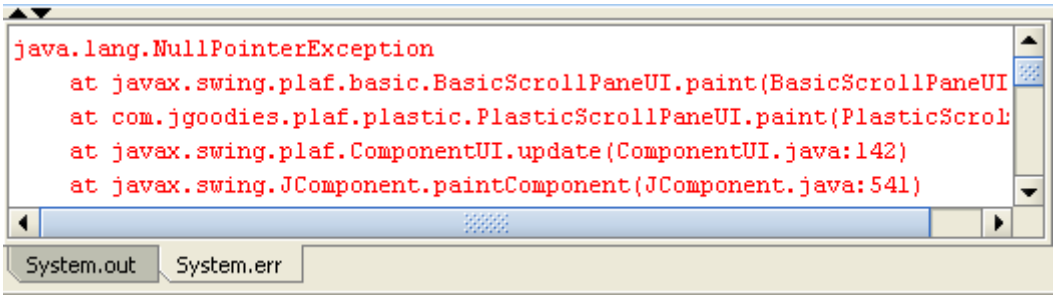


Abbildung 5.9: Ausgabeströme

Die zwei Textfelder sind durch Reiter getrennt. Sollte eine Ausgabe in dem nicht angezeigten Reiter erfolgen, beginnt dieser zu blinken, um anzuzeigen, dass sich sein Inhalt geändert hat.

Statusleiste Eine schnelle Übersicht über den Zustand der Umgebung liefert die Statusleiste, die in Abbildung 5.10 dargestellt wird. Hier finden sich Informationen darüber, aus wievielen Prozessen ( ) der gerade geladene Algorithmus

besteht, wieviele Nachrichten schon ausgeliefert wurden ( ) oder wieviele sich in der Messagequeue befinden ( ). Außerdem wird der aktuelle Zeitpunkt angezeigt, der sich in Zeit ( ) und Zeitschritt ( ) unterteilt. Zeitschritte werden allerdings nur für den asynchronen Ablauf der Simulation verwendet. Dort erfolgt die Darstellung von Nachrichten, die zur selben Zeit empfangen werden, sequenziell. Der Zeitpunkt des Eintreffens ist zwar weiterhin der gleiche, jedoch wird dieser Zeitpunkt nochmal in einzelne Zeitschritte geteilt. Was es mit den Zeitschritten genau auf sich hat, wird in Abschnitt 5.3.4 näher erläutert.

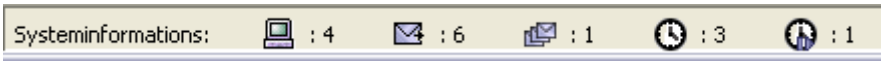


Abbildung 5.10: Statusleiste

5.3.3 Einstellungen

Das „Options"-Menü (Abbildung 5.11) enthält eine Auswahl an Einstellungen, mit denen die Simulation konfiguriert wird.

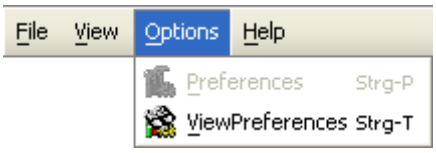


Abbildung 5.11: Einstellungen

- Preferences : Konfiguration des Algorithmus.
- ViewPrefereces : Anpassungen der Ansichten.

Preferences Die hier vorgestellten Einstellungen dienen dazu, ein verteiltes System zu erstellen oder es im Nachhinein zu bearbeiten, in dem ein verteilter Algorithmus agieren kann. Sowohl bei der Erstellung einer neuen Umgebung ( ), als auch beim Aufruf der Einstellungen ( ), wird ein Dialog sichtbar, der sich in zwei Kategorien unterteilt:

1. Agents : hier werden die Agenten, die für den Algorithmus benötigt werden, eingebunden und initialisiert.

2. Topology : hier können die vorher erstellten Agenten positioniert und Verbindungen zwischen den einzelnen Agenten erstellt werden.

In der Kategorie „Agents“ werden die einzelnen Teilnehmer eines Algorithmus verwaltet (Abbildung 5.12).

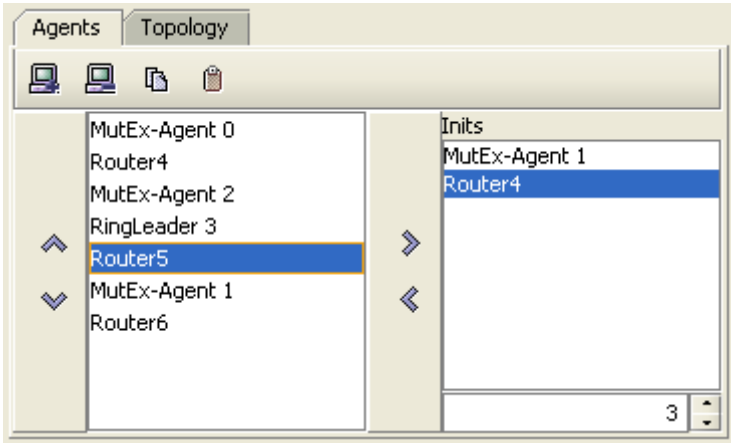


Abbildung 5.12: Einstellungen (Agenten)

Für diesen Zweck gibt es eine Reihe von Befehlen:

- Add Agent : öffnet einen Dialog, der die Auswahl einer .class-Datei ermöglicht. Von der ausgewählten Klasse wird eine neue Instanz erzeugt, sofern es sich bei der Klasse um eine Erweiterung der `Agent`-Klasse handelt. Der daraus entstandene Agent wird dann in das System integriert.
- Remove Agent : mit diesem Befehl werden die ausgewählten Agenten wieder aus dem System entfernt.
- Copy : die Klasse des ausgewählten Agenten wird in eine interne Zwischenablage (unabhängig zur Systemzwischenablage) kopiert.
- Paste : fügt eine neue Instanz der Agentenklasse aus der internen Zwischenablage ein. Diese Funktionalität ist vor allem dann praktisch, wenn für einen Algorithmus mehrere Objekte einer Klasse verwendet werden.

Die daraus entstandene Liste kann weiterhin bearbeitet werden. Die Reihenfolge kann über ⌃ und ⌄ geändert werden. Diese Reihenfolge könnte bei einigen Topologiemanagern eine Rolle spielen. Im Falle des TokenRing-Topologiemanagers wird z.B. jeder Agent mit dem in der Liste folgenden verbunden.

Um einen Algorithmus starten zu können, muss mindestens ein Agent initialisiert werden. Die Initialisierung erfolgt über eine `InitMessage`, die als eine erste auslösende Aktion verstanden werden kann. Eine solche Nachricht kann für jeden in der Liste ausgewählten Agenten über » der Messagequeue und der Liste, die diese Nachrichten anzeigt, hinzugefügt werden. Aus dieser Liste können die Nachrichten aber auch wieder mit « entfernt werden. Unterhalb der Liste mit den Initialisierungsnachrichten kann eine Verzögerungszeit angegeben werden, falls einige Agenten zu einem späteren Zeitpunkt starten sollen. Es ist auch möglich, einen Agenten mehrfach zu initialisieren.

Hinter dem „Topology"-Reiter (Abbildung 5.13) befinden sich die eben erstellten Agenten wieder. Diese stehen allerdings noch für sich allein und müssen nun mit den anderen Teilnehmern verbunden werden. Zusätzlich können hier die Positionen der einzelnen Prozesse festgelegt werden.

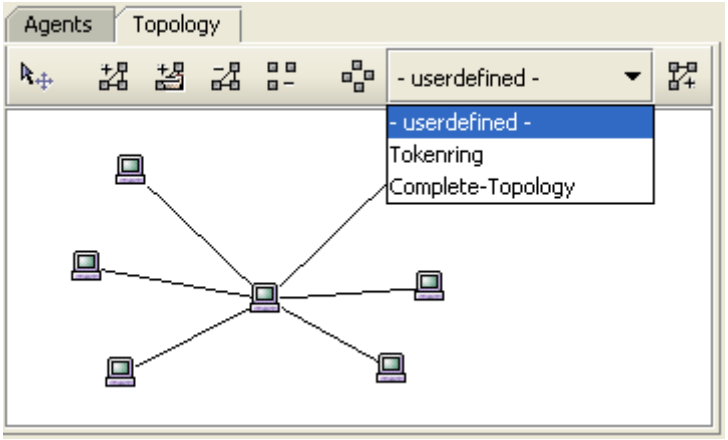


Abbildung 5.13: Einstellungen (Topology)

Zur Bearbeitung der Topologie werden folgende Werkzeuge angeboten:

- Move : nach Auswahl dieses Werkzeugs können die Positionen einzelner Agenten verschoben werden.
- Connect : mit diesem Werkzeug können Verbindungen zwischen zwei Agenten gezogen werden.
- Connect from file : dieses Werkzeug funktioniert ähnlich zu dem vorangegangenen, mit dem Unterschied, dass hier eine Klasse gewählt werden kann, aus der neue `Connection`-Instanzen erzeugt werden können. Auf diesem Weg können eigene Verbindungen (siehe Abschnitt 5.2.3) eingebunden werden.

- Disconnect : löscht die Verbindung, auf die geklickt wurde.
- Remove all connections : hebt alle Verbindungen wieder auf.
- position the agents : stellt die vom Topologiemanager festgelegten Positionen wieder her.

Topologiemanager übernehmen die lästige Aufgabe, Agenten miteinander zu verbinden. Zusätzlich legen sie die Anordnung der Agenten fest. Es besteht die Auswahl zwischen verschiedenen Topologiemanagern:

1. -userdefined- : diese Auswahl zeigt an, dass die Verbindungen und Positionen manuell erstellt worden sind oder noch erstellt werden müssen. Wird die von einem Topologiemanager erstellte Umgebung geändert, z.B. durch Löschen einer Verbindung, wird automatisch diese Auswahl aktiv. Durch einen Eingriff in die Arbeit eines Topologiemanagers verzichtet der Benutzer auf dessen weitere Untestützung.
2. TokenRing : der Tokenring ordnet die Agenten in einem Ring an und verbindet jeweils die benachbarten Agenten miteinander. Die dabei verwendeten Verbindungen lassen nur eine unidirektionale Kommunikation zu.
3. Complete-Topology : dieser Topologiemanager verbindet alle Agenten miteinander.

Wenn eigene Regeln für das Verbinden der Agenten benutzt werden sollen, können mit eigene Topologiemanager ausgewählt und eingebunden werden. Beim allgemeinen Umgang mit Topologiemanagern sei zu beachten, dass bei der Wahl eines neuen Topologiemanagers alle bisherigen Einstellungen, sowohl Verbindungen als auch Positionen, verloren gehen und durch die des neuen ersetzt werden.

Alle hier getätigten Änderungen sowohl im Agenten- als auch im Topologiebereich können wieder verworfen werden („Abbrechen“). Erst mit „OK“ werden alle Änderungen wirkungsvoll.

ViewPreferences In diesem Abschnitt wird erklärt, wie vorhandene Ansichten angepasst werden, indem zusätzliche Renderer hinzugefügt oder bestehende gelöscht werden oder wie die Renderer angepasst werden. Es wird auch gezeigt, welche Schritte notwendig sind, um eine eigene Ansicht in das Programm zu integrieren. Bei den folgenden Änderungen sei zu beachten, dass diese, im Gegensatz zu den Einstellung des vorangegangenen Abschnitts, sofort wirkungsvoll werden und und nicht mehr verworfen, sondern nur rückgängig gemacht werden können.

Abbildung 5.14 zeigt die Oberfläche, die das Bearbeiten der Ansichten ermöglicht und die die Befehle beinhaltet, die für diesen Zweck zur Verfügung stehen.

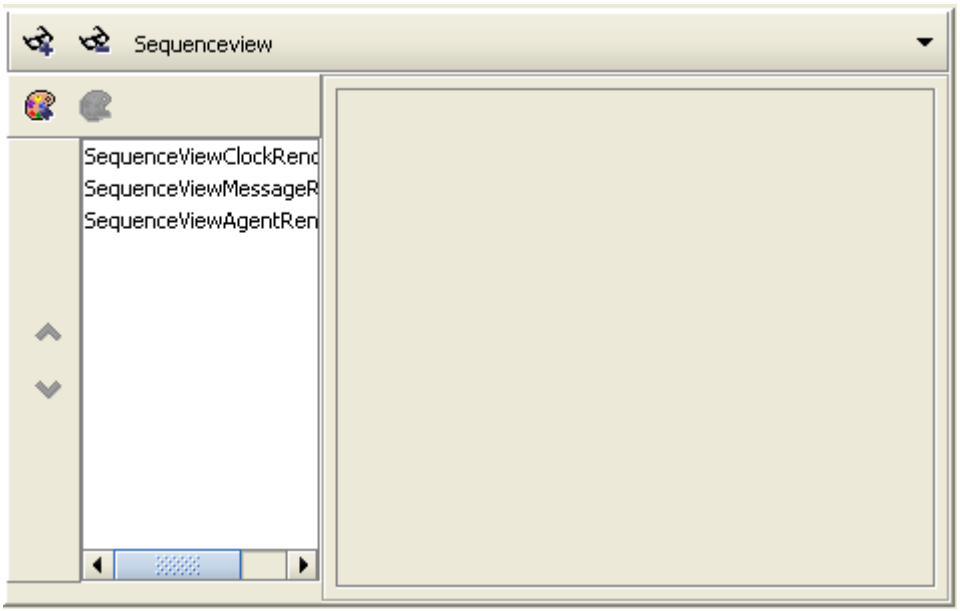


Abbildung 5.14: Einstellungen (View)

- Add View : fügt dem System eine neue Ansicht hinzu. Hierzu wird eine Klasse ausgewählt, die die Klasse View erweitert. Genaueres über das Erstellen eigener Ansichten ist in Abschnitt 5.2.4 zu lesen.
- Remove View : entfernt die gerade ausgewählte Ansicht.
- Add Renderer : erweitert die aktuelle Ansicht um eine Instanz der ausgewählten Renderer-Klasse.
- Remove Renderer : entfernt die ausgewählten Renderer.
- Up : setzt den ausgewählten Renderer eine Position in der Liste nach oben.
- Down : setzt den ausgewählten Renderer eine Position nach unten. Beim Erreichen der Listengrenzen wird der Renderer einfach am anderen Ende angehängt. Die Reihenfolge der Renderer in dieser Liste bestimmt auch die Reihenfolge, in der die Renderer ausgeführt werden. Dies ist nicht unwichtig, da ein Renderer Teile des vor ihm agierenden Renderers übermalen kann.

Eigene Renderer zu implementieren ist mit unter sehr zeitaufwändig, besonders, wenn hierbei nur kleine Änderungen an der Darstellung vorgenommen werden sollen. Aus diesem Grund gibt es die Möglichkeit, die bereits bestehenden

Renderer zu bearbeiten und dabei ihr Aussehen zu verändern. Hierbei handelt es sich zwar oft nur um kleinere Einstellungen wie Farbe oder Form, aber genau diese Änderungen machen schon viel aus. Diese Einstellungen werden beim Auswählen eines Renderers sichtbar, sofern dieser die Möglichkeit bietet. Die Einstellmöglichkeiten der bereits vorhandenen Renderer werden in den folgenden Abschnitten beschrieben.

Zunächst werden die Renderer vorgestellt, die für das Zeichnen der Agenten zuständig sind. Diese bestehen sowohl in der Topologie- als auch in der Ablaufansicht aus einer Liste von weiteren Renderern, und zwar für jeden Agenten einer. Abbildung 5.15 zeigt eine Tabelle, die zum einen die einzelnen Agenten des System und zum anderen den für diesen Agenten verwendeten Renderer. Es

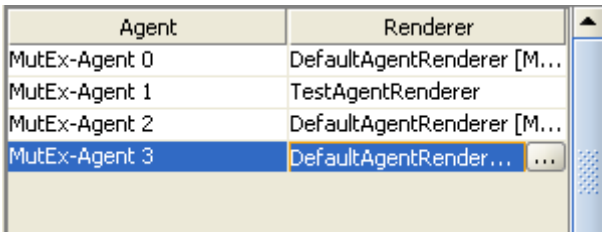


Abbildung 5.15: Renderereinstellungen (Agentenliste)

besteht die Möglichkeit, den Renderer eines speziellen Agenten auszutauschen, indem eine neue Klasse ausgewählt wird, die diese Aufgabe übernimmt. Nach Auswahl einer Zeile wird ein Knopf sichtbar (mit „...“ beschriftet), der eine Dateiauswahl aufruft. Über diese wird die Klasse des neuen Renderers ausgewählt. Ein Doppelklick auf eine Zeile führt den Anwender direkt zu den Einstellungen des gewählten Renderers.

Die Einstellungsmöglichkeiten der einzelnen Renderer können bei dieser Gelegenheit ebenfalls vorgestellt werden. Abbildung 5.16 zeigt den Dialog, über den dies geschieht. Der Dialog zeigt die möglichen Zustände, die der gewählte Agent

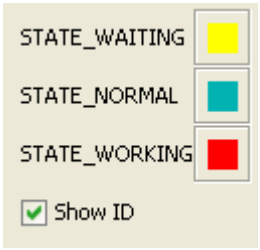


Abbildung 5.16: Renderereinstellungen (Agenten)

annehmen kann und in welcher Farbe dieser Zustand in der gewählten Visualisierung dargestellt wird. Nach einem Mausklick auf ein solches Farbfeld wird ein Dialog sichtbar, über den eine andere Farbe für diesen Zustand festgelegt werden kann. Zusätzlich kann verhindert werden, dass die ID des entsprechenden Agenten in der Visualisierung angezeigt wird. Die hier gemachten Einstellungen werden allerdings nur bei diesem Agenten und nur in dieser Ansicht aktiv. Wenn das veränderte Farbschema ebenso bei allen anderen Agenten verwendet werden soll, muss der Anwender die Änderungen bei jedem Agenten einzeln durchführen oder eigene Renderer oder eigene Ansichten verwenden.

Neben den Agenten können auch die Nachrichten angepasst werden. Der hierfür zuständige Dialog wird in Abbildung 5.17 gezeigt. Für jeden bereits exis-

Abbildung 5.17: Renderereinstellungen (Nachrichten)

tierenden Nachrichtentyp kann, wie zuvor bei den Zuständen der Agenten, eine Farbe gewählt werden. Auch hier werden die Einstellungen nur für die aktuelle Ansicht wirkungsvoll.

Der ConnectionRenderer ist nur in der Topologieansicht zu finden. Dieser hat die Aufgabe, die Verbindungen zwischen den Agenten zu zeichnen. Verbindungen werden durch Linien dargestellt. Die Farbe und das Aussehen dieser Linien kann hier eingestellt werden. Das Aussehen der Linie unterscheidet sich in ihrer Breite oder ob sie gestrichelt oder durchgezogen sind. Der hierfür verantwortliche Dialog wird in Abbildung 5.18 dargestellt.

5.3.4 Ablauf und Steuerung

Da ein Algorithmus in endlich vielen Schritten abläuft, stellt DisASTer eine Steuerung bereit, die der eines Video- oder CD-Spielers ähnelt (Abbildung 5.19). Abgesehen von dem automatischen Ablauf des Algorithmus besteht die Möglichkeit, den Algorithmus in Einzelschritten ablaufen zu lassen und diese Schritte sogar wieder rückgängig zu machen. Diese Rückwärtsbewegung erinnert an die Verlaufsfunktion eines Browser.

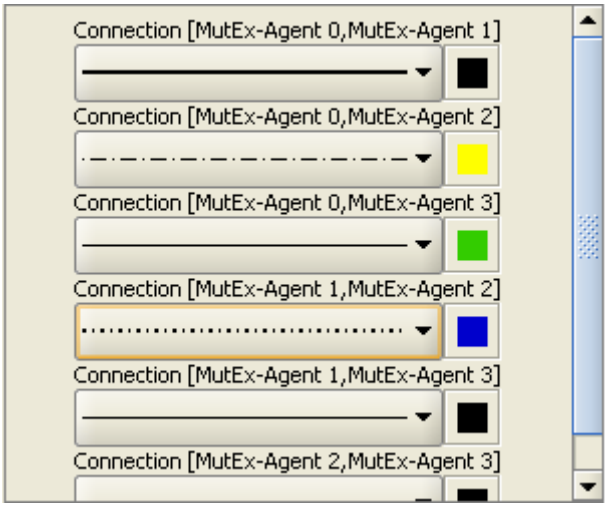

Connection [MutEx-Agent 0,MutEx-Agent 1]
Connection [MutEx-Agent 0,MutEx-Agent 2]
Connection [MutEx-Agent 0,MutEx-Agent 3]
Connection [MutEx-Agent 1,MutEx-Agent 2]
Connection [MutEx-Agent 1,MutEx-Agent 3]
Connection [MutEx-Agent 2,MutEx-Agent 3]

Abbildung 5.18: Renderereinstellungen (Verbindungen)

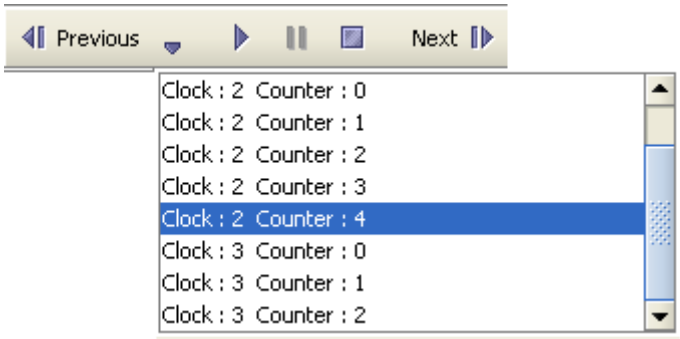

Previous
Next
Clock : 2 Counter : 0
Clock : 2 Counter : 1
Clock : 2 Counter : 2
Clock : 2 Counter : 3
Clock : 2 Counter : 4
Clock : 3 Counter : 0
Clock : 3 Counter : 1
Clock : 3 Counter : 2

Abbildung 5.19: Steuerung

Der Ablauf selbst kann ebenfalls angepasst werden. Der Benutzer kann eine Geschwindigkeit einstellen, mit der er die Simulation gerne ablaufen lassen möchte, wobei er entscheiden kann, ob die Visualisierung dieser Simulation nun animiert werden soll oder nicht. Neben diesen Einstellungen kann der Benutzer hier den Simulationsmodus wählen und entscheiden, ob die Simulation manuell gesteuert werden soll oder von einem der beiden automatischen Ablaufmodi. Abbildung 5.20 zeigt die Oberfläche, die diese Einstellungen ermöglicht. Die manuelle Steuerung erlaubt dem Benutzer selbst in den Ablauf des Algorithmus einzugreifen. Auf diese Weise kann das nicht deterministische Verhalten von verteilten Systemen simuliert oder diverse Speziallfälle durchgespielt werden.

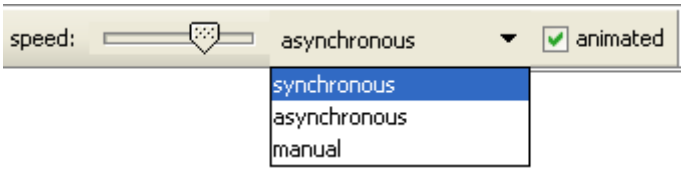


Abbildung 5.20: Steuerungsmodi

Steuerung Zunächst soll aber auf die einfache, automatische Steuerung der Simulation eingegangen werden.

- Play : mit diesem Befehl wird der automatische Ablauf eines Algorithmus gestartet oder wieder aufgenommen. Dieser wird nun schrittweise im eingestellten Tempo ausgeführt.
- Pause : der automatische Ablauf kann mit diesem Befehl angehalten werden. Der aktuelle Schritt wird dabei allerdings noch zu Ende geführt.
- Stop : anders als der Pause-Befehl hält dieser Befehl den Ablauf nicht an, sondern bricht ihn regelrecht ab und setzt den Algorithmus auf seinen Anfangszustand zurück.
- Previous : mit diesem Befehl kann der letzte Zustand eines Algorithmus wieder hergestellt werden. Diese Funtionalität erinnert an den Verlauf eines Webbrowsers. Wie dort können auch hier mehrere Schritte auf einmal rückgängig gemacht werden, in dem der gewünschten Zeitpunkt aus einer Liste gewählt wird.
- Next : führt einen Einzelschritt durch, sofern der Algorithmus durch oder unterbrochen ist. Mit Hilfe des Einzelschritts kann der Anwender die Geschwindigkeit der Simulation selbst betimmen. Diese Methode ist jedoch nicht mit der manuellen Steuerung zu vergleichen, da in diesem Fall der Ablauf schon feststeht. Im Falle der manuellen Steuerung kann der Anwender noch in den Ablauf des Algorithmus eingreifen.

Steuerungsmodi Die Einstellungsmöglichkeiten, die für den Ablauf eines Algorithmus zur Verfügung stehen, werden in diesem Abschnitt näher erläutert. Dabei wird unter anderem auf die einzelnen Modi, die für die Steuerung vorgesehen sind, eingegangen.

Die Geschwindigkeit wird durch einen Schieberegler bestimmt und zwar von langsam (ganz links) bis schnell (ganz rechts). Dadurch wird die Zeitspanne, die für einen Schritt benötigt wird, verändert.

Um einen Algorithmus in DisASTer ablaufen zu lassen, bestehen drei Möglichkeiten:

1. synchronous : bedeutet, dass alle Nachrichten, die zu einem Zeitpunkt ausgeliefert werden sollen, auch zeitgleich ausgeliefert werden. Diese Einstellung stellt das normale Verhalten des Algorithmus dar.

2. asynchronous : bedeutet, dass Nachrichten, die zum selben Zeitpunkt ausgeliefert werden sollen, zwar zum selben Zeitpunkt, nicht aber im selben Zeitschritt ausgeliefert werden. Ein Zeitpunkt kann aus mehreren Zeitschritten bestehen. Pro Zeitschritt kann nur eine Nachricht ausgeliefert werden. Beim synchronen Ablauf werden alle Zeitschritte parallel ausgeführt. In dieser Einstellung geschieht dies, der Übersichtlichkeit halber, nacheinander. Für den Algorithmus selber macht diese Einstellung keinen Unterschied.

3. manuell : mit dieser Einstellung kann auf „Handbetrieb“ geschaltet werden, um den Ablauf des Algorithmus individuell zu bestimmen.

Das Umschalten zwischen den einzelnen Modi ist jederzeit möglich. Das ist von Vorteil, wenn der Ablauf eines Algorithmus nur an einer bestimmten Stelle durch ein manuelles Eingreifen geändert werden soll. Nach diesem Eingriff kann die Simulation wieder automatisch erfolgen.

In der Regel laufen die Simulationen animiert ab. Vor allem bei Verwendung der Topologieansicht ist dies sinnvoll, um die Kommunikation zwischen den einzelnen Prozessen zu beobachten. Sollte es aber unerwarteter Weise zu Leistungseinbußungen kommen, können die Animationen auch ausgeschaltet werden.

Manuelle Steuerung Der automatische Ablauf eines Algorithmus kann oft sehr praktisch sein. Wie bereits oben beschrieben, ist die Möglichkeit gegeben, die Simulation auf Handsteuerung umzuschalten. Diese Art der Steuerung geschieht über die AgentMessageQueues, die nach Auswahl des manuellen Modus sichtbar werden. Abbildung 5.8 auf Seite 59 zeigt bereits eine solche AgentMessageQueue.

Funktionen der AgentMessageQueues sind:

- Next : ähnlich zur automatischen Steuerung sorgt dieser Befehl dafür, dass die *oberste* Nachricht in der Liste ausgeliefert wird. Es spielt dabei auch keine Rolle, ob ein späterer Zeitpunkt für das Ausliefern dieser Nachricht vorgesehen ist. Das ist eine Möglichkeit den Ablauf des Algorithmus durcheinander zu bringen.

- Delete : mit diesem Befehl wird die ausgewählte Nachricht gelöscht. Dieser Schritt kann durch auch wieder rückgängig gemacht werden.

- Up : um die Nachrichtenreihenfolge zu ändern - denn es wird immer zuerst die *oberste* Nachricht ausgeliefert - können Nachrichten nach oben oder

- Down : nach unten verschoben werden. Wird die Ober- bzw. Untergrenze der Liste erreicht, wird die Nachricht am anderen Ende eingereiht. Allerdings muss hier beachtet werden, dass das Ändern der Reihenfolge in einer AgentMessageQueue keinen Einfluß auf die Reihenfolge der Nachrichten in der eigentlichen Messagequeue hat. Wird die Nachrichtenreihenfolge im manuellen Modus geändert und dann wieder in den automatischen gewechselt, läuft die Simulation so ab wie vor der Änderung.

6 Entwurf

Dieses Kapitel beschäftigt sich mit dem objektorientierten Entwurf der zu erstellenden Anwendung. Hierzu wird zunächst eine Analyse verteilter Algorithmen durchgeführt, um deren allgemeine Eigenschaften bestimmen zu können. Anschließend wird mit diesem Hintergrundwissen der Entwurf beschrieben, der sich in vier Teile untergliedert: dem Modell, den Ansichten, der Kontrollinstanz und der Grafischen Oberfläche.

6.1 Eigenschaften verteilter Algorithmen

Neben der Recherche anderer Projekte (Kapitel 9) und deren Herangehensweise an die vorliegende Problematik, wird eine eigene Analyse durchgeführt, bei der die allgemeinen Eigenschaften verteilter Algorithmen untersucht werden. Ziel dabei ist es, die Eigenschaften zu finden, die nahezu alle verteilten Algorithmen in sich vereinen. Unter Berücksichtigung dieser Eigenschaften soll das Programm später in der Lage sein, möglichst alle verteilten Algorithmen simulieren zu können.

Ganz allgemein lässt sich hier festhalten, dass verteilte Algorithmen aus mehreren Teilnehmern bestehen. Diese besitzen meist einen bestimmten Zustand der z.B. anzeigt, dass sie gerade beschäftigt sind oder sie auf eine Bestätigung der anderen Teilnehmer warten, um eine bestimmte Aktion auszuführen. Des weiteren können die Teilnehmer eine bestimmte Rolle annehmen, die sie von den anderen Teilnehmern unterscheidet. Ein Teilnehmer könnte z.B. als Anführer der anderen fungieren oder als zentraler Server agieren.

Ein weiterer Punkt, den alle verteilten Algorithmen in sich vereinen, ist die Kommunikation untereinander. Dies ist auch nicht weiter verwunderlich, da verteilte Algorithmen ausschließlich in verteilten Systemen wie Netzwerken eingesetzt werden. Diese Kommunikation besteht aus Nachrichten, die sich die Teilnehmer untereinander senden. Das Eintreffen einer Nachricht löst dann beim Empfänger eine bestimmte Aktion aus, bei der sich z.B. der Zustand des Empfängers ändert. Dieser wiederum kann dann den anderen Teilnehmern die Änderung über weitere Nachrichten zukommen lassen, auf die diese ebenfalls reagieren können.

Mit diesen beiden Eigenschaften, den einzelnen Teilnehmern mit ihren Zuständen und Rollen, sowie der einfachen Kommunikation zwischen ihnen über Nachrichten, können nahezu alle verteilten Algorithmen abgebildet werden. Im folgenden Abschnitt werden nun diese Erkenntnisse mit denen der anderen Projekte zusammengefasst, um einen möglichst allgemeinen und vielfältig einsetzbaren

Entwurf zu entwickeln.

6.2 Erkenntnisse für den eigenen Entwurf

Im Folgenden werden die Ergebnisse der vorangegangenen Analyse und die ähnlicher Projekte (Kapitel 9) verwendet, um den eigenen Entwurf zu spezifizieren. Als erstes wird die Sprache festgelegt. Das fertige Programm sollte möglichst schnell eingesetzt werden können und leicht zu bedienen sein, damit sich der Anwender ganz auf die Implementierung der Algorithmen konzentrieren kann. Um das zu garantieren, wird keine neue Sprache entwickelt, sondern eine bereits bekannte eingesetzt. Hier spricht vieles für Java. Denn Java bietet eine objektorientierte, plattformunabhängige und freie Lösung, die zusätzlich von einer breiten Masse verwendet werden kann. Gerade die Objektorientierung erlaubt eine einfache und logische Handhabung, denn alle Elemente in dem simulierten Netzwerk sind als Objekt greifbar.

Nun gilt, es den eigentlichen Algorithmus zu definieren. Dieser kann wie bei *DAJ* oder *ViSiDiA* global festgelegt werden, indem alle Aktionen in einer Klasse spezifiziert werden. Die Logik des Algorithmus kann aber auch auf die einzelnen Teilnehmer verteilt werden, indem jeder dieser Teilnehmer entscheidet, wann er warum welche Aktion wie durchführt. In diesem Fall fällt die Entscheidung für den zweiten, dezentralen Ansatz, da dieser zunächst einfacher zu handhaben und verständlicher zu sein scheint. Er lässt sich besser mit der Realität vergleichen, in der jeder Netzwerkteilnehmer selbst entscheidet, wie er sich verhält. Hinzu kommt, dass die meisten Algorithmen in Pseudocode definiert werden können, der das Verhalten der einzelnen Teilnehmer beschreibt.

Als nächstes wird der Simulationstyp bestimmt. Zur Auswahl stehen die ereignisgesteuerte oder die prozessorientierte Simulation (siehe Kapitel 9). Eine Unterart der ereignisgesteuerten ist die nachrichtengesteuerte Simulation. Dieser Ansatz wird in diesem Entwurf weiterverfolgt, da sich die einzelnen Teilnehmer untereinander über Nachrichten verständigen, deren Eintreffen als Ereignis eingestuft werden kann. Anstatt eine Liste von Ereignissen zu verwalten, wird eine Nachrichtenliste, die MessageQueue, verwendet. Diese enthält im Verlauf einer Simulation alle gesendeten Nachrichten bevor diese dem eigentlichen Adressaten zugestellt werden, anstatt sie direkt an diesen auszuliefern. Mit Hilfe dieser Zwischenstation ist es nun möglich, in den Verlauf des Algorithmus einzugreifen und diesen gegebenfalls zu ändern, indem die MessageQueue bearbeitet wird.

Um eine Kommunikation zwischen den einzelnen Teilnehmern zu ermöglichen, müssen diese miteinander verbunden werden. Dieser Verbund bildet schließlich

einen Graphen, bestehend aus Knoten und Kanten, nämlich den Teilnehmern und ihren Verbindungen. Dieser Graph beschreibt die Topologie des nun entstandenen verteilten Systems. Eine solche Topologie soll sich ebenfalls im Entwurf wiederfinden. Mit ihr können die Positionen der einzelnen Prozesse bestimmt werden. Dadurch bekommt das System ein Gesicht und der Anwender hat die Topologie, die Positionen der einzelnen Prozesse sowie deren Verbindung grafisch vor Augen.

Die Visualisierung stellt den schwierigsten und zugleich wichtigsten Teil des Programms dar. Für die Visualisierung wurden bereits zwei Techniken aufgeführt. Zum einen kann die Visualisierung zeitgleich mit der Simulation ablaufen. Zum anderen kann die Simulation eine Log-Datei erzeugen, aus der die Visualisierung ihre Informationen nimmt; dies hat den Vorteil, dass die Visualisierung unabhängig von der Simulation betrachtet werden kann, allerdings ist hierbei an ein Eingreifen in das Geschehen nicht zu denken. Aus diesem Grund soll die Visualisierung zeitgleich mit der Simulation ablaufen, da ja die Möglichkeit, den Ablauf eines Algorithmus zu ändern, mit der MessageQueue gegeben ist. Und wenn schon ein Eingreifen möglich ist, wäre es auch interessant, Teile der Simulation unter veränderten Bedingungen zu wiederholen, ohne gleich den ganzen Algorithmus von neuem zu starten. Diese Funktionalität kann aber nur gegeben werden, wenn das Programm weiß, was in den vorherigen Schritten passiert ist. Um dies zu erreichen, soll der Verlauf der Simulation gespeichert werden. Auf diese Weise kann der Benutzer wie in einem Webbrowser an eine beliebige Stelle des Algorithmus zurückspringen, um ihn entweder einfach nur zu wiederholen oder ihn unter veränderten Bedingungen weiterzuführen.

Ein letzter Punkt betrifft die Steuerung der Simulation. Zunächst wäre hier ein automatischer Ablauf zu betrachten. Dieser könnte, ähnlich einem Video- oder CD-Spieler, mit den Funktionen Start, Stopp, Pause und Einzelschritt bestückt sein. Durch einen automatischen Ablauf fällt aber zunächst die Eigenschaft der MessageQueue, verändert werden zu können, unter den Tisch. Durch eine zusätzliche manuelle Steuerung wird dies wieder möglich. Auf diesem Weg kann die Nachrichtenreihenfolge geändert oder es können sogar Nachrichten gelöscht werden. Der oben beschriebene Eingriff in den Verlauf der Simulation fällt ebenfalls in den Aufgabenbereich der Steuerung.

Die folgenden Abschnitte stellen den eigentlichen Entwurf dar. Dieser wurde nach einem erweiterten MVC-Modell entwickelt. Das MVC-Paradigma - Model, View, Controller - wird sehr häufig in objektorientierten Entwürfen verwendet. Es unterteilt eine Anwendung in drei Teilstrukturen: das Modell, das i.d.R. dem Datenmodell entspricht, eine oder mehrere Ansichten, die dieses Modell grafisch darstellt und eine oder mehrere Kontrollinstanzen oder Modifikatoren, mit deren Hilfe das Datenmodell verändert werden kann (vgl. [Gamma et. al. 96], S. 4ff). Um alle drei Komponenten untereinander konsistent zu halten, wird häufig das

Entwurfsmuster „Observer“ eingesetzt, wodurch eine Änderung des Modells den anderen Komponenten mitgeteilt werden kann. Eine Beschreibung dieses Entwurfsmusters findet sich ebenfalls bei der „Gang of Four“ (vgl. [Gamma et. al. 96], S. 257ff). In diesem Entwurf findet sich eine vierte Komponente, die Grafische Benutzeroberfläche (GUI) des Programms. Da die anderen drei Komponenten möglichst unabhängig voneinander arbeiten sollen, und das soweit möglich ohne eine grafische Benutzeroberfläche, wurde eine vierte hinzugefügt. Diese hat Zugriff auf die anderen drei Komponenten. In der ursprünglichen Idee des MVC-Musters würde die Aufgabe der grafischen Benutzeroberfläche auf die Ansichten zurückfallen. Da für dieses Programm aber die Visualisierung eine wichtige Rolle spielt, wurde sie getrennt von der übrigen grafischen Oberfläche betrachtet und entworfen. Abbildung 6.1 zeigt den Aufbau der hier verwendeten Variante des MVC-Paradigmas.

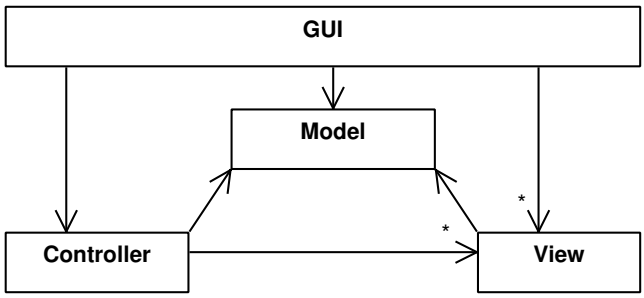


Abbildung 6.1: MVC-Modell

Aus dem Diagramm wird ersichtlich, dass sowohl Controller als auch die Views (mehrere sind möglich) auf die Daten des Modells zugreifen. Sobald der Controller eine Änderung am Modell vorgenommen hat, benachrichtigt er die Ansichten und nicht wie üblich das Modell. Diese Vorgehensweise hängt mit der Art der Visualisierung zusammen, da der Controller zusätzlich für die Regelung der Animationen zuständig ist. Die Änderung des Modells findet bereits vor dem Ablauf einer Animation statt und würden folglich nur einmal den Ansichten mitgeteilt werden. Diese sollen sich aber während einer Animation mehrfach aktualisieren. Da der Controller weiss, wann und in welchen Abständen das Neuzeichnen erfolgen soll, übernimmt er die Aufgabe, die Ansichten zu benachrichtigen, die dann den Zustand des Modells grafisch darstellen können. Auf diese drei Komponenten setzt schließlich die GUI auf, die den Zugriff und die Steuerung aller anderen Komponenten besitzt. In der Regel würde die grafische Oberfläche unter die Ansichten fallen, aber in diesem Fall übernimmt sie doch einige Aufgaben mehr, als nur die Darstellung des Modells. Über sie wird das Modell gespeichert oder geladen, der Controller angesteuert und die Ansichten eingebettet.

6.3 Model

Dieser Abschnitt beschreibt das Datenmodell der Anwendung. Kern des Modells ist das `Environment`. Das `Environment` verwaltet alle einzelnen Teile des Systems, wie in Abbildung 6.2 dargestellt wird. Zu diesen Teilen zählen die Topologie, die das Netzwerk beschreibt, in dem der Algorithmus eingesetzt wird, die MessageQueue, die als Ereignisliste angesehen werden kann, der Verlauf, der alle bereits abgearbeiteten Ereignisse und Aktionen speichert, sowie das Agentlog, in dem die Textausgaben der Teilnehmer gespeichert und verwaltet werden. Ausserdem werden alle gesendeten Nachrichten in einer zusätzlichen Liste verwaltet. Das `Environment` selbst wird als „Singleton“ implementiert, d.h. es gibt nur eine Instanz dieser Klasse, auf die über die statische Methode `getInstance()` zugegriffen werden kann. Durch den Einsatz dieses Entwurfsmusters wird die Einmaligkeit des Datenmodells sichergestellt (vgl. [Gamma et. al. 96], S. 139ff).

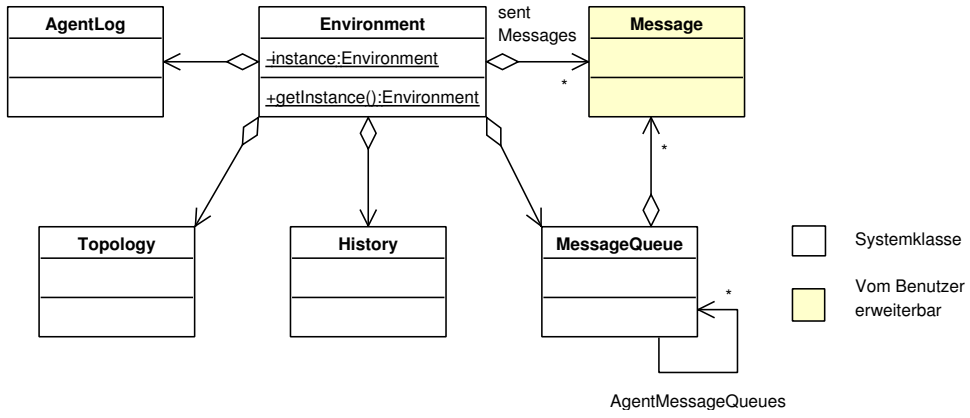


Abbildung 6.2: Aufbau der Umgebung

Abbildung 6.3 zeigt die verwendeten Observer, die bei Änderungen des Modells benachrichtigt werden. Diese Observer werden als Schnittstellen (Interfaces) definiert. Aus Effizienzgründen werden zwei verschiedene verwendet. Der `EnvironmentChangeListener` bekommt jede Änderung des Modells mit, wo hingegen der `EnvironmentUpdateListener` nur benachrichtigt wird, wenn die Umgebung neu geladen wurde. Ein Neuladen der Umgebung tritt nicht nur dann auf, wenn der Benutzer tatsächlich eine neue Umgebung erzeugt oder eine vorher gespeicherte geladen wird, sondern auch, wenn ein im Verlauf enthaltener Zustand wiederhergestellt wird.

Die Topologie beschreibt den Aufbau des zu simulierenden verteilten Systems. Hier werden alle Teilnehmer des Systems und dessen Verbindungen untereinander

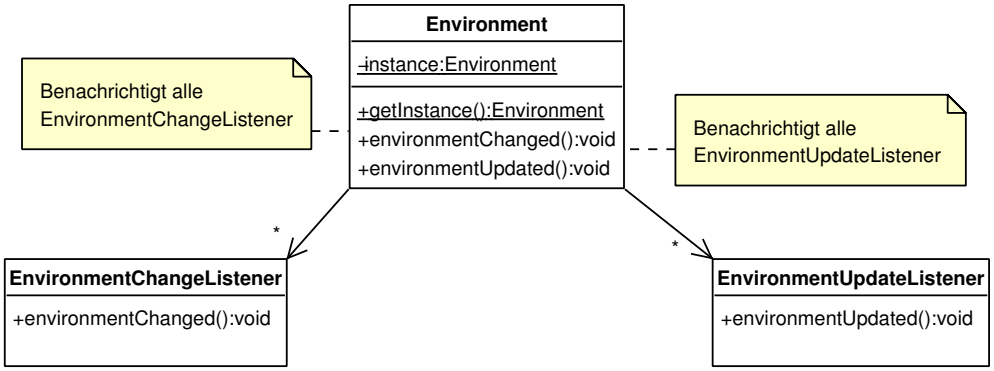


Abbildung 6.3: Observermodell des Environments

erzeugt und verwaltet. Die Teilnehmer dieses Systems werden Agenten genannt. Die Topologie wird als Graph dargestellt, wobei die Agenten die Knoten und die Verbindungen die Kanten beschreiben. Die Verwaltung der Agenten, ihre Positionen und Verbindungen, kann durch Topologiemanger erfolgen. Eine Topologie selbst kann eine Reihe solcher Manager besitzen. Der Anwender kann dann entscheiden, ob er einen dieser Topologiemanager verwenden will oder ob er die Positionierung und die Verbindungen manuell einstellt. Wenn ein Manager ausgewählt wird, übernimmt dieser beim Hinzufügen oder Entfernen von Agenten die Aufgabe, diese zu positionieren und Verbindungen mit den anderen Teilnehmern zu setzen. Eine mögliche Konstellation, die aus der Arbeit eines Topologiemanagers entstehen kann, ist z.B. ein Verbundring, in dem nur die benachbarten Agenten miteinander verbunden werden, ähnlich einem Token Ring. Eine weitere Möglichkeit wäre es, alle Agenten untereinander zu verbinden. Um eine hohe Erweiterbarkeit des Systems zu garantieren, können diese Elemente, Topologiemanager (`TopologyManager`), Verbindungen (`Connection`) oder Agenten (`Agent`), vom Benutzer erweitert werden. Abbildung 6.4 zeigt den Aufbau der Topologie, bei der bereits die zwei vorgestellten Manager aufgeführt werden.

Die MessageQueue ist neben der Topologie und dessen Agenten ein weiterer wichtiger Bestandteil der Simulationsumgebung. Sie übernimmt die Nachrichtensteuerung. Alle Ereignisse werden in Form von Nachrichten aufgefasst. Dazu gehören nicht nur die Nachrichten, die sich die Agenten untereinander schicken, sondern auch jene, die ein Agent sich selbst schickt oder die, die er vom System erhält. Bevor ein Agent eine gesendete Nachricht erhält, wird diese zunächst in die MessageQueue aufgenommen. Ab diesem Zeitpunkt übernimmt die Kontrollinstanz (siehe Abschnitt 6.5) die Entscheidung, wann eine Nachricht an den

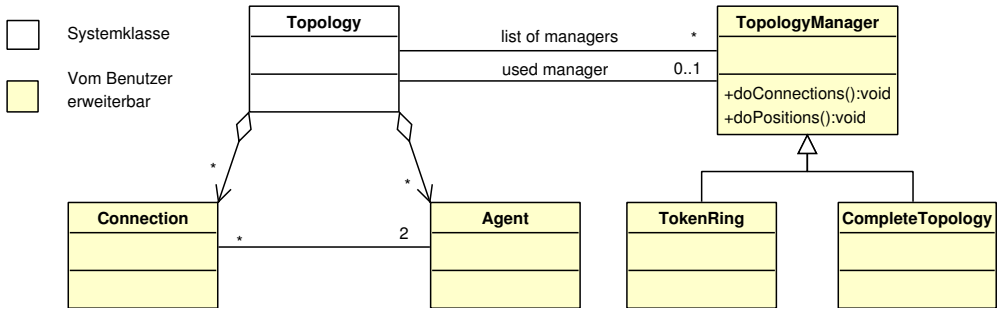


Abbildung 6.4: Aufbau der Topologie

entsprechenden Empfänger ausgeliefert wird. Die MessageQueue dient als globale Nachrichten- oder Ereignisliste. Daneben verwaltet sie aber auch eine eigene MessageQueue für jeden Agenten. In diesen finden sich ausschließlich die Nachrichten, die für diesen Agenten bestimmt sind. Die AgentMessageQueues beinhalten keine weiteren MessageQueues. Wird die Simulation vom Anwender selbst gesteuert (siehe Abschnitt 6.5), hat dieser die Möglichkeit, diese Listen zu verändern, indem Nachrichten gelöscht werden oder deren Reihenfolge durcheinandergebracht wird. Gesendete Nachrichten werden aber nicht aus dem System entfernt, sondern finden sich in einer gesonderten Liste, die das Environment selbst verwaltet, wieder. Abbildung 6.5 zeigt den Aufbau der MessageQueue.

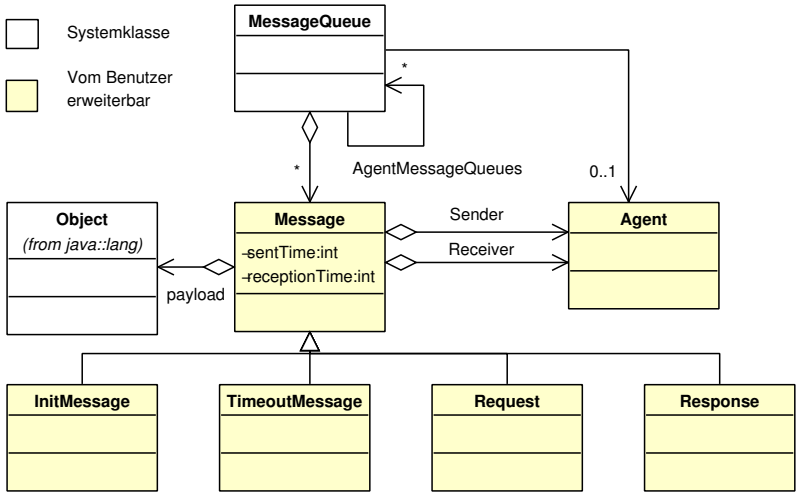


Abbildung 6.5: Aufbau der MessageQueue

In der Abbildung wird ebenfalls der Aufbau einzelner Nachrichten ersichtlich. Diese setzen sich aus folgenden Elementen zusammen:

- `sentTime` : legt fest, an welchem Zeitpunkt die Nachricht gesendet wurde.
- `receptionTime` : hat zwei Aufgaben. Nachdem die Nachricht beim Empfänger angekommen ist, zeigt dieser Wert den Zeitpunkt des Eintreffens an. Bevor die Nachricht jedoch die MessageQueue verlässt, findet sich hier die Zeit, zu der die Nachricht ausgeliefert werden soll. Wenn dieser Wert über dem aktuellen Zeitpunkt liegt, bleibt die Nachricht zunächst in der MessageQueue liegen, bis dieser Zeitpunkt erreicht ist. Auf diesem Weg ist es möglich, Nachrichten mit einer bestimmten Zeitverzögerung auszuliefern.
- `payload` : repräsentiert die eigentliche „Fracht“, also die Informationen, die mit einer Nachricht übermittelt werden sollen.
- `sender` : nimmt den Wert des Senders an. Falls eine Nachricht vom System gesendet wurde, was bei den Initialisierungen der Fall ist, ist dieser Wert null.
- `receiver` : hat den Wert des Empfängers. Dieser Wert kann auch die Broadcast-Adresse (`Agent.BROADCAST`) annehmen, wodurch Kopien der Nachricht an alle erreichbaren Nachbarn des Senders weitergeleitet werden.

Damit die Nachrichten unterschiedliche Ereignisse auslösen können, werden verschiedene Typen verwendet, die noch zusätzlich erweitert werden können. Es soll möglich sein jeden Nachrichtentyp in einer separaten Methode abzufangen und diesen entsprechend zu behandeln. Das ist allerdings Aufgabe des Agenten, der jetzt genauer beschrieben wird.

Die Rolle der Agenten wird im Entwurf gesondert betrachtet, da der Anwender durch sie die verteilten Algorithmen implementiert, indem er die Agentenklasse erweitert. Dabei wird das Verhalten eines Prozesses in einem bestimmten Algorithmus definiert. Damit stellen Agenten das wichtigste Werkzeug für diese Aufgabe dar. Aus diesem Grund erhalten sie eine Reihe von Eigenschaften und Methoden, auf die zurückgegriffen werden kann, um möglichst einfach und schnell einen verteilten Algorithmus für das vorliegende System zu entwickeln. Die Struktur der Agenten wird in Abbildung 6.6 dargestellt.

Im folgenden werden die Eigenschaften der Agenten aufgelistet:

- `state` : Variable, die den Zustand des Agenten beschreibt. Mögliche Zustände können „beschäftigt“, „wartend“ oder „normal“ sein. Sollten diese Zustände nicht ausreichen, kann ein Agent um weitere Zustände erweitert werden.
- `role` : Variable, welche die Rolle des Agenten beschreibt. Die Rolle kann vor allem im Vergleich mit anderen Agenten ausschlaggebend sein. Mögliche

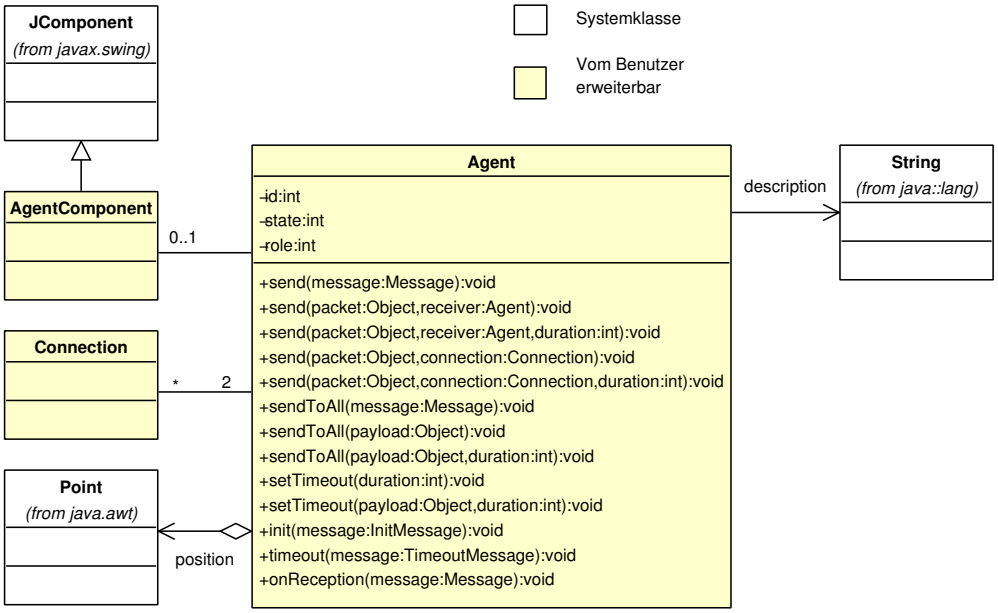


Abbildung 6.6: Aufbau der Agenten

Rollen sind „Master“ oder „Slave“. Diese beiden Werte können auch unter mehreren Bedingungen eingesetzt werden. Die Rolle „Master“ kann z.B. für den Anführer oder einen zentralen Server eines Systems gelten.

- `position` : beschreibt die Position des Agenten in seiner Topologie.
- `id` : ist ein eindeutiger Zahlenwert, der als Unterscheidungskriterium verwendet wird. Besonders wenn für einen Algorithmus nur ein einziger Agententyp verwendet wird, ist die ID der einzige Unterschied zwischen den Agenten. Zudem kann mit den IDs eine Hierarchie unter den Agenten erzeugt werden, wobei z.B. eine niedrigere ID hierarchisch unter einer höheren steht.
- `connections` : beinhaltet alle Verbindungen zu anderen Teilnehmern. Hierfür wird eine Hashtabelle verwendet, in der die Verbindung zu einem Agenten mit ihm als Schlüssel gespeichert wird. Dadurch wird zusätzlich verhindert, dass ein Agent mehrere Verbindungen zu einem anderen aufnehmen kann.
- `agentComponent` : Swingkomponente, durch die der Anwender grafische Informationen und zusätzliche Befehle zu seinen Agenten hinzufügen kann.

- `description` : stellt eine einfache, textuelle Beschreibung des jeweiligen Agenten dar.

Zu allen aufgeführten Eigenschaften gibt es entsprechende Zugriffsmethoden (`getXXX()`, `setXXX()`), um mit diesen Werten auch außerhalb der Klasse Agent arbeiten zu können. Daneben finden sich noch eine Reihe weiterer nützlicher Methoden, die im Folgenden aufgeführt werden. Um die Nachrichtensteuerung zu verwenden, bietet die Klasse `Agent` einige Methoden zum Senden von Nachrichten:

- `send(Message message)` : sendet die übergebene Nachricht an den oder die eingetragenen Empfänger. Durch diese Methode können auch unterschiedliche Nachrichtentypen versendet und damit verschiedene Ereignisse ausgelöst werden.

- `send(Object payload, Agent receiver)` : mit dieser Methode wird ein Objekt an einen bestimmten Empfänger gesendet. Hierbei wird der Standardnachrichtentyp (`disaster.model.Message`) verwendet.

- `send(Object payload, Agent receiver, int duration)` : diese Methode sendet ebenfalls eine Standardnachricht an den übergebenen Empfänger, allerdings wird diese Nachricht frühestens nach der angegebenen Zeitverzögerung ausgeliefert. Der übergebene Wert wird relativ zum Zeitpunkt des Sendens gesehen.

- `send(Object payload, Connection connection)` : mit dieser Methode wird ein Agent nicht direkt, sondern über seine Verbindung angesprochen.

- `send(Object payload, Connection connection, int duration)` : diese Methode funktioniert wie die vorherige, mit dem Unterschied, dass hier wieder eine Zeitverzögerung mit angegeben werden kann.

- `sendToAll(Message message)` : mit dieser Methode wird eine Art Broadcast simuliert. Alle über bestehende Verbindungen mit dem aktuellen Agenten erreichbaren Teilnehmer erhalten eine Kopie der übergebenen Nachricht. Das Übergeben eines Empfängers an eine Nachricht, die über diese Methode versendet wird, hat keine Wirkung. Ist der übergebene Empfänger mit dem Sender dieser Nachricht verbunden, bekommt er, wie alle anderen auch, eine Kopie dieser Nachricht. Neben dieser Methode könnte auch eine einfache sende-Methode verwendet werden, wobei als Empfänger die Broadcastadresse, `Agent.BROADCAST` angegeben wird. Das Verhalten unterscheidet sich nicht. Es wird ebenfalls eine Nachrichtenkopie für jeden erreichbaren Agenten erzeugt.

- `sendToAll(Object payload)` : über diese Methode kann ein Agent ein bestimmtes Paket an alle seine erreichbaren Nachbarn schicken. Es wird wiederum der Standardnachrichtentyp verwendet.

- `sendToAll(Object payload, int duration)` : bei dieser Methode wird erneut eine Zeitverzögerung angegeben.

Wenn ein Agent ein Ereignis bei sich selbst auslösen möchte, kann er natürlich auf die obigen Methoden zurückgreifen, allerdings wurde für diesen speziellen Fall ein weiterer Mechanismus, der Timeout, eingeführt. Um einen solchen Timeout zu erzeugen, gibt es zwei zusätzliche Methoden:

- `setTimeout(int duration)` : mit dieser sendet der Agent eine Timeoutnachricht an sich selbst. Diese wird nach der angegebenen Zeit ausgeliefert. Dadurch kann eine geplante Aktion zu einem späteren Zeitpunkt ausgeführt und der entsprechende Agent schlafen gelegt werden. Dieser Mechanismus sollte als Ersatz für `Thread.sleep()` verwendet werden, denn `Thread.sleep()` bezieht sich auf die reale Zeit, wo hingegen ein Timeout die Simulationszeit verwendet. Allerdings „schläft“ der entsprechende Agent nicht wirklich. In der Zeit, in der ein solcher Timeout abläuft, können weitere Nachrichten eintreffen und der Agent weitere Aktionen durchführen. Dem Anwender bleibt an dieser Stelle überlassen, ob und wie er solche Aktionen während eines Timeouts unterbindet. Auf den Einsatz von Threads sollte grundsätzlich verzichtet werden, da die Agenten keine Thread-Sicherheit implementiert haben und eine Blockierung des laufenden Threads durch z.B. `Thread.sleep()` das ganze System lahmlegen kann. Dieses Programmierverhalten erinnert an die Handhabung von Swing-Elementen, speziell dem `ActionListener`, die von dem

  ```
  EventDispatchingThread
  ```

  verwaltet werden und dieser ebenfalls nicht unterbrochen werden sollte.

- `setTimeout(Object payload, int duration)` : sendet erneut einen Timeout. Hierbei kann allerdings noch zusätzlich ein Objekt verschickt werden, dass mit dem Timeout ausgeliefert wird.

Neben dem Senden ist vor allem das Empfangen von Nachrichten eine wichtige Funktion der Agenten. Um bei jedem Nachrichtentyp eine andere Methode aufrufen zu können und damit besser zwischen den einzelnen Ereignissen unterscheiden zu können, wird an dieser Stelle ein Pseudopolymorphismus verwendet. Eine Nachricht wird über die Methode `onReception(<Nachrichtentyp>)` empfangen. Gibt es für den übergebenen Typ keine entsprechende Methode, wird eine Methode gesucht, die die Oberklasse dieses Typs empfangen kann. Diese

Suche wird solange fortgesetzt bis der Standardnachrichtentyp `Message` erreicht ist und für diesen Typ stellt bereits die Klasse `Agent` eine Methode bereit. Mit dieser Technik ist es nun möglich, unterschiedliche Aktionen bei unterschiedlichen Ereignissen (Nachrichtentypen) auszuführen. Es werden zwei Spezialfälle mit entsprechenden Methoden behandelt:

- `init(InitMessage message)` : wird beim Empfang einer Initialisierungsnachricht aufgerufen. Welcher Agent wann initialisiert wird, legt der Benutzer bei der Einrichtung des verteilten Systems fest. Ein Agent kann auch mehrfach initialisiert werden. Um einen Algorithmus starten zu können, muss allerdings mindestens ein Agent auf diese Weise angestossen werden.

- `timeout(TimeoutMessage message)` : wird nach Ablauf eines Timeouts aufgerufen.

`println(Object out)` ist eine weitere wichtige Methode, die dem Benutzer zur Verfügung steht. Diese Methode soll die Standardausgabe ersetzen. Das übergebene Objekt wird in einer separaten Konsole ausgegeben und zusätzlich mit gespeichert, denn die hier getätigten Ausgaben finden sich im AgentLog wieder, der im nächsten Abschnitt beschrieben wird.

Das AgentLog ist eine Liste, in der alle über `println()` gemachten Ausgaben eingehen. Für jeden Aufruf wird ein neuer Eintrag (`AgentLogEntry`) angelegt, der aus einem Text und dem Agenten, der die Ausgabe gemacht hat, besteht. Auch hier wird wieder mit Observern gearbeitet, die benachrichtigt werden, sobald ein neuer Eintrag in das Log eingetragen wird. Der Aufbau des AgentLog findet sich in Abbildung 6.7.

Die Speicherung der einzelnen Umgebungszustände in einem Verlauf (siehe Abbildung 6.8) wird ähnlich wie das Agentlog behandelt. Es handelt sich dabei um eine Liste aus Einträgen, die aus der Zeit, zu der sie erstellt wurden, und dem derzeitigen Zustand des Systems bestehen. Ein solcher Zustand wird mit Hilfe der Klasse `EnvironmentState` serialisiert. Diese zusätzliche Klasse ist notwendig, da das Environment selbst nicht serialisiert werden kann und auch nicht soll. Durch den häufigen Einsatz von Observern müssten diese ebenfalls serialisiert werden. Dadurch würde die Dateigröße der abgespeicherten Umgebungen nur unnötig anwachsen. Teilweise sind hier auch Klassen betroffen, die nicht so einfach bzw. gar nicht serialisiert werden können, da es sich um nicht serialisierbare Klassen aus dem Javasytem handelt. Aus diesem Grund speichert die Klasse `EnvironmentState` alle notwendigen Komponenten des Modells, wie die MessageQueue, die Topologie usw. Über diese Klasse wird ebenfalls ein kompletter Algrithmus abgespeichert, der dann in eine Datei serialisiert wird. Um

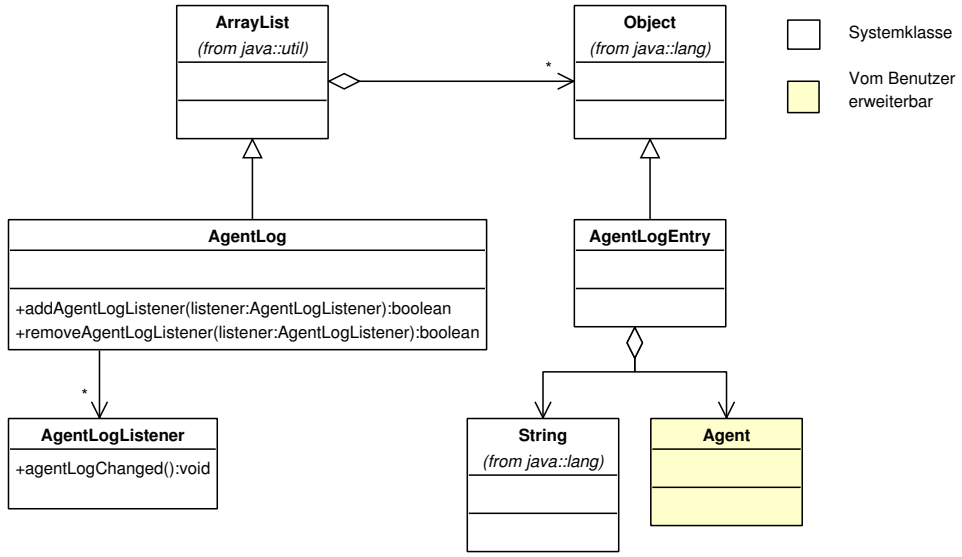


Abbildung 6.7: Aufbau des Agentlogs

einen gespeicherten Zustand, sei es aus dem Verlauf heraus oder aus einer Datei, wieder herstellen zu können, wird die `updateEnvironment()`-Methode eines `EnvironmentState`-Objekts aufgerufen, die das Environment mit den gespeicherten Werten aktualisiert.

Die Zeit der Simulation und damit des Verlaufs teilt sich in Zeit bzw. Zeitpunkt (`clock`) und Zeitschritt (`timestep`), wobei ein Zeitpunkt aus mehreren Zeitschritten besteht. Was es genau damit auf sich hat, wird in Abschnitt 6.5 beschrieben.

6.4 View

Die Ansichten (Views) der Anwendung werden für die Visualisierung der Algorithmen verwendet. Um sie zu realisieren, wird das Entwurfsmuster „Brücke“ verwendet. Dieses Entwurfsmuster macht eine Entkopplung einer Abstraktion und ihren Implementierung, um beide unabhängig voneinander zu variieren (vgl. [Gamma et. al. 96], S. 165ff). In diesem Entwurf wird sogar eine „Mehrfach-Brücke“ verwendet, wobei die Abstraktion auf mehrere Implementierungen zurückgreift. Die Verbindung zwischen diesen Elementen wird als Brücke bezeichnet. Die Abstraktion wird durch die Klasse `View` repräsentiert. Dabei handelt es sich um eine Erweiterung einer `JComponent` aus der Swing-Bibliothek. Auf dieser Komponente

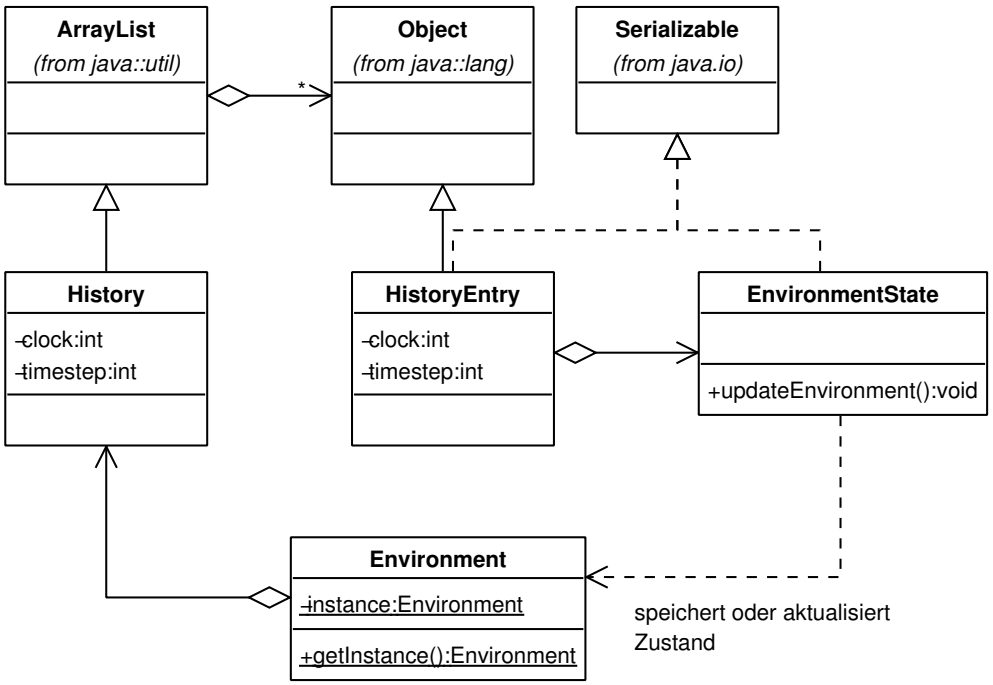


Abbildung 6.8: Aufbau des Verlaufs

soll die Visualisierung stattfinden. Die Zeichenoperationen selbst werden von den Implementierungen, den `Renderern`, durchgeführt, von denen eine `View` mehrere besitzen kann. Wird diese Komponente dann mittels `paintComponent()` neu gezeichnet, wird von allen enthaltenen Renderern die `draw()`-Methode aufgerufen, in der die eigentlichen Darstellungen definiert werden. Der Aufbau der Ansicht und der darin enthaltenen Brücke, die zwischen `View` und `Renderer` geschlagen wird, wird in Abbildung 6.9 gezeigt.

Die `Renderer` erhalten das `Graphics`-Objekt der `View` und verwenden diese als Zeichenfläche. Einen Spezialfall stellen die `AgentRenderer` dar. Diese besitzen eine Referenz auf einen Agenten und werden von einem weiteren Renderer verwaltet. Für jeden Agenten kann ein solcher `AgentRenderer` verwendet werden. Der Ansicht sollte aber nur der Renderer bekannt sein, der diese verwaltet. Innerhalb dessen Zeichenoperation, werden die der einzelnen `AgentRenderer` aufgerufen. Diese Vorgehensweise zeigt, dass Renderer auch beliebig tief geschachtelt werden können, wenn z.B. diese `AgentRenderer` erneut mehrere Renderer beinhalten.

Die Visualisierung sollte zudem animiert ablaufen können. Eine Animation wird in der Regel wie ein Film erzeugt, bei dem in einer bestimmten Zeit mehrere Bilder hintereinander angezeigt werden, wodurch dem Betrachter eine Bewegung vorgetäuscht wird. Diese Methode wird bei den Ansichten ebenfalls verwendet.

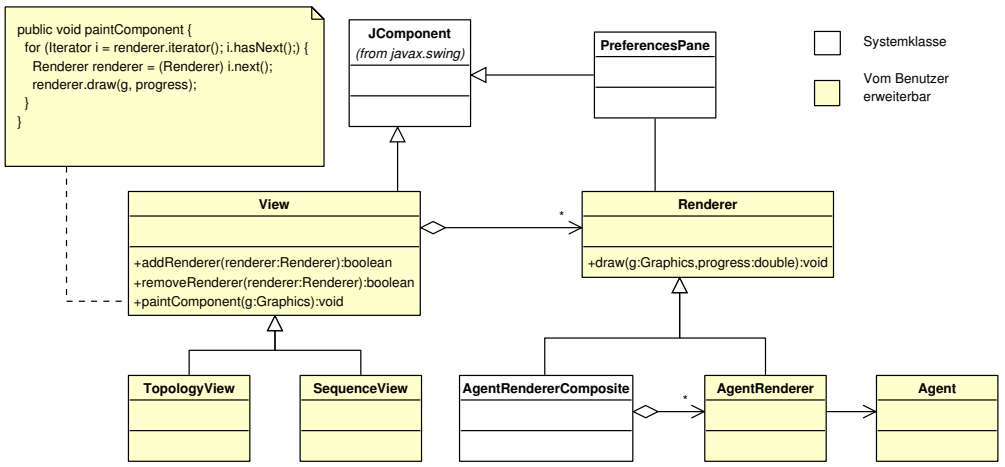


Abbildung 6.9: Aufbau der Ansicht

Aus diesem Grund besitzt die Zeichenmethode der `Renderer` einen Parameter, der den prozentualen Fortschritt eines solchen Animationsschritts angibt. Dieser Parameter besitzt einen Wert von 0.0, für 0% bis 1.0 für 100%. Falls der Anwender auf Animationen verzichten will, z.B. aus Effizienzgründen, beträgt dieser Wert immer 1.0, um den finalen Zustand der Animation darzustellen. Ein `Renderer` ist, falls er nur für eine statische Darstellung verantwortlich ist, nicht gezwungen, auf diesen Parameter zurückzugreifen.

6.5 Controller

Der Controller hat die Aufgabe, die Simulation zu steuern. Für diesen Zweck wurden zwei bzw. drei verschiedene Modi gewählt. Zum einen die manuelle Steuerung, bei welcher der Anwender den Ablauf des Algorithmus weitgehend selbst bestimmen kann. Er kann entscheiden, welche Nachrichten wann ausgeliefert werden oder ob er Nachrichten aus dem System entfernen möchte. Neben der manuellen Steuerung stehen ihm noch zwei automatische zur Verfügung. Hier wird zwischen synchron und asynchron unterschieden. Der synchrone Ablauf entspricht dem wirklichen Ablauf. Hier wird jeder Zeitpunkt nacheinander ausgeführt. Beim asynchronen Ablauf wird ein solcher Zeitpunkt in Zeitschritte aufgeteilt. Pro Nachricht wird ein Einzelschritt dargestellt. Dieser Modus erleichtert die Betrachtung eines Algorithmus, wenn viele Nachrichten zur selben Zeit über die Leitungen wandern. Logisch gesehen geschieht das dann zwar immer

noch, aber die Darstellung wird in kleinere Schritte geteilt. Der automatische Ablauf soll wie ein Video- oder CD-Player gesteuert werden können. Funktionen wie Start, Stop, Pause, Einzelschritt oder Rückschritt sollen diese Arbeitsweise wiedergeben, wobei das Rückwärtslaufen eher mit dem Verlauf eines Webbrowsers verglichen werden sollte. Die Stop-Funktion soll zum einen die Simulation anhalten und zum anderen den Startzustand des Algorithmus wiederherstellen. Ein Umschalten zwischen den drei Modi ist jeder Zeit möglich. Dadurch kann der Anwender z.B. einen Großteil der Simulation automatisch ablaufen lassen und nur an Stellen eingreifen, die für ihn interessant sind.

Abbildung 6.10 zeigt den Aufbau des Controllers. Dieser wurde mit dem Entwurfmuster „State“ (Zustand) realisiert. Mit Hilfe dieses Entwurfsmusters kann ein Objekt seinen internen Zustand ändern, so dass es so aussieht, als ob das Objekt seine Klasse geändert hat (vgl. [Gamma et. al. 96], S. 357ff). Es werden aber nur die internen Zustände durch jeweils andere Objekte repräsentiert.

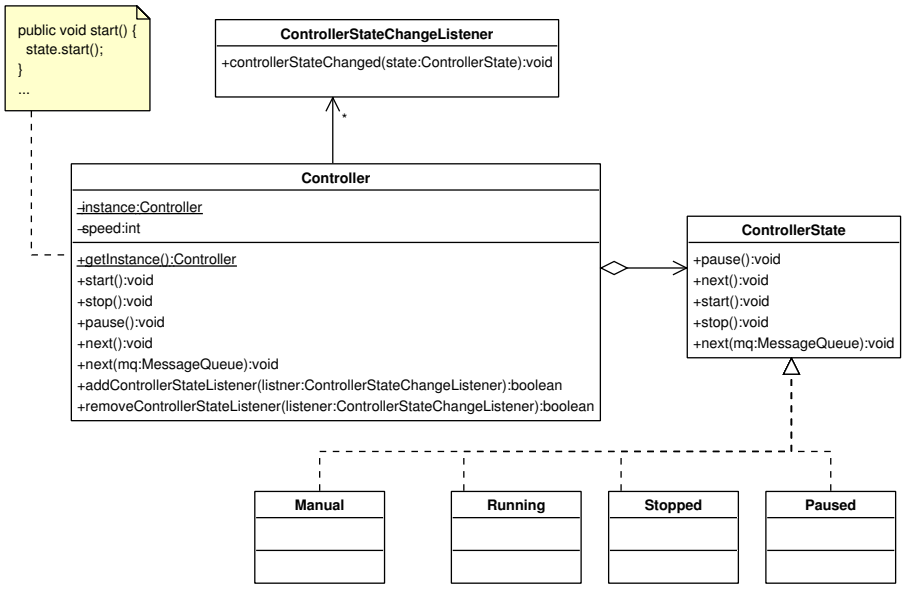


Abbildung 6.10: Aufbau des Controllers

Der Controller verfügt über folgende Zustände, die alle das Interface `ControllerState` implementieren:

- `Manual` : dieser Zustand wird angenommen, wenn die Simulation vom Benutzer selbst gesteuert wird.
- `Running` : beschreibt den Zustand, wenn die Simulation automatisch abläuft, unabhängig davon, ob dies synchron oder asynchron, animiert oder

nicht animiert geschieht. Die Geschwindigkeit dieses Ablaufs kann verändert werden.

- `Paused` : wird angenommen wenn die Simulation unterbrochen wird.
- `Stopped` : bezeichnet den Zustand, der zu Beginn einer jeden Simulation steht.

Die Arbeitsweise dieser Zustände wird im Zustandsdiagramm der Abbildung 6.11 sichtbar.

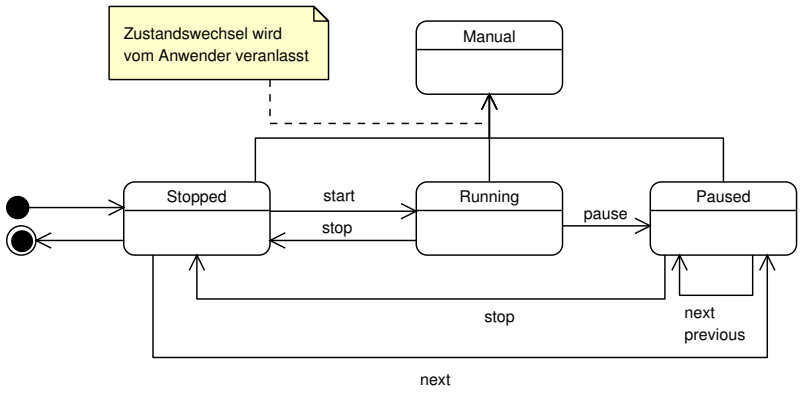


Abbildung 6.11: Zustandswechsel des Controllers

Zusätzlich findet sich nochmals das Entwurfsmuster „Observer“ im Controller wieder. Beobachter des Controllers werden bei jedem Zustandswechsel benachrichtigt und bekommen den neuen Zustand mitgeteilt. Von dieser Technik wird vor allem bei den Knöpfen der GUI Gebrauch gemacht, die diese Zustandswechsel hervorrufen sollen; denn diese Knöpfe sollen nicht jederzeit bedient werden können. So wäre es beispielsweise sinnlos den Pausenknopf betätigen zu können, wenn sich der Controller schon im Pausenzustand befindet.

Um nur eine einzige Kontrollinstanz zu garantieren, steht auch der Controller nur als Singleton zur Verfügung. Dieses Entwurfsmuster wurde bereits beim Modell in Abschnitt 6.3 vorgestellt.

6.6 Grafische Benutzeroberfläche (GUI)

Die grafische Benutzeroberfläche (engl.: Grafical User Interface = GUI) stellt die Schnittstelle zum Benutzer her. Sie bietet dem Anwender Interaktionsmöglichkeiten, um eine Umgebung zu erstellen, zu ändern, sie zu speichern oder zu laden,

sie zu steuern und vor allem sie zu beobachten. Um all das zu ermöglichen, greift die GUI sowohl auf das Modell als auch auf den Modifikator und die Ansichten zu. Wie aus Abbildung 6.12 ersichtlich wird, bildet die GUI eine Baumstruktur. Die Wurzel stellt die Klasse `MainFrame` dar. Die aus `JFrame` abgeleitete Klasse beinhaltet weitere Komponenten und Container, aus denen schließlich die Anwendung entsteht. Für diese Struktur wird erneut das Entwurfsmuster „Composite“ verwendet. Damit für jeden Programmaufruf nur ein solches Hauptfenster sichtbar wird, wurde dieses ebenfalls als Singleton realisiert.

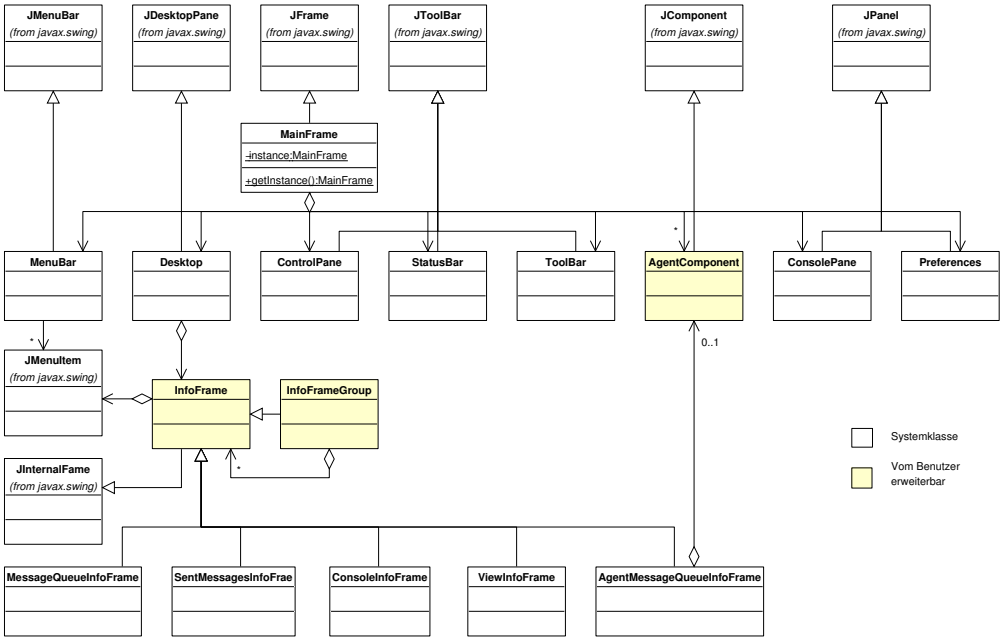


Abbildung 6.12: Aufbau der grafischen Oberfläche

Wesentliche Bestandteile des Hauptfensters sind:

- `MenuBar` : das Menü, in dem sich alle wichtigen Menüeinträge, wie das Laden oder Speichern der Umgebung, der Aufruf der Informationsfenster oder der Einstellungen sowie der Online-Hilfe, befinden. Da einige der hier befindlichen Befehle auch über die ToolBar zugänglich sind, werden sie als Singleton bereitgestellt.

- `ControlPane` : besitzt alle Knöpfe, die für das Steuern der Simulation benötigt werden. Darunter fallen die Knöpfe um die Simulation zu starten, anzuhalten, zu pausieren, sie mittels Einzelschritt ablaufen zu lassen oder im Verlauf zurück zu gehen. Desweiteren kann hier die Geschwindigkeit und die Simulationsart eingestellt werden.

- `StatusBar` : diese schmale Leiste am unteren Rand des Fensters der Anwendung gibt einen kurzen Überblick über den Zustand der Umgebung. Zu diesen Informationen gehören die Anzahl der Agenten, der Nachrichten in der MessageQueue, die Anzahl der bereits ausgelieferten Nachrichten oder die aktuelle Simulationszeit.
- `ToolBar` : bietet eine Auswahl der wichtigsten Befehle des Hauptmenüs. Hierzu gehören Laden, Speichern oder der Aufruf der Einstellungen.
- `Preferences` : sind Oberflächen, um die Simulationsumgebung einzurichten und zu bearbeiten sowie die Ansichten anzupassen.
- `AgentComponent` : diese wurden bereits in Abschnitt 6.3 im Abschnitt über die Agenten vorgestellt. Sie werden vom Hauptfenster verwaltet und in den AgentMessageQueue-Fenstern dargestellt, die später erläutert werden.
- `ConsolePane` : beinhaltet die Standardausgabe (`System.out`) und die Standardfehlerausgabe (`System.err`).
- `Desktop` : diese Komponente nimmt den Großteil der Anwendung in Anspruch. Auf dem Desktop finden sich viele weitere Fenster, die wichtige Informationen beinhalten. Der Inhalt dieser Fenster wird im folgenden Abschnitt vorgestellt.

Auf dem `Desktop` finden sich Unterklassen der Klasse `JInternalFrame`, die sog. `InfoFrames`. Sie besitzen einen Menüeintrag (`JMenuItem`), das dem Hauptmenü (`MenuBar`) der Anwendug übergeben wird, um dieses Fenster aufzurufen. Falls mehrere Fenster zusammen gehören, können diese in eine `InfoFrameGroup` aufgenommen werden. Eine solche Gruppe wird als ein Untermenü im Hauptmenü aufgeführt, wobei alle Fenster dieser Gruppe einzeln oder gemeinsam aufgerufen und dargestellt werden können. Zu den Informationsfenstern gehören:

- `MessageQueueInfoFrame` : in dem die MessageQueue in Tabellenform abgebildet wird.
- `SentMessagesInfoFrame` : in diesem Fenster werden alle bereits ausgelieferten Nachrichten dargestellt. Die Darstellung erfolgt in einer ähnlichen Weise wie die der MessageQueue in einer Tabelle.
- `ConsoleInfoFrame` : zeigt das Agentlog an.
- `ViewInfoFrame` : in solchen Fenstern werden die Ansichten dargestellt. Die Fenster der Ansichten werden in einer `InfoFrameGroup` und in einem entsprechenden Untermenü der Menüleiste verwaltet. Die vom Anwender hinzugefügten Ansichten finden sich dann ebenfalls in dieser Gruppe wieder.

- `AgentMesssageQueueInfoFrames` : hier befinden sich jeweils die Nachrichten, die an einen bestimmten Agenten adressiert wurden. Zusätzlich werden in diesen Fenstern die `AgentComponents` dargestellt. Auch diese Fenster bilden eine eigene Gruppe.

Die genaue Funktionsweise der einzelnen Oberflächenelemente wird im Benutzer-Handbuch (Kapitel 5) beschrieben.

7 Implementierung und Test

Im vorangegangenen Kapitel wurde der Entwurf des Programms vorgestellt. Auf den folgenden Seiten wird auf ein paar besondere Fälle eingegangen, die in der Implementierungsphase aufgetreten sind. Hierbei handelt es sich, neben ein paar besonders kniffligen Programmteilen, um verwendete Werkzeuge oder Bibliotheken. Abschließend wird auf die Tests des Programms eingegangen.

7.1 Topologieeditor

Der Topologieeditor ist ein Teil der Einstellungen und wird dazu verwendet, die einzelnen Agenten der Umgebung zu positionieren und Verbindungen zwischen ihnen zu erzeugen. Die Arbeitsweise des Topologieeditors ähnelt der eines Grafikprogramms und wird auch wie ein solches implementiert. Zur Umsetzung wird aus diesem Grund das Entwurfsmuster „State“ verwendet. Die Struktur und die Vorgehensweise dieses Entwurfsmusters wurde bereits im Entwurf der Kontrollinstanz, Abschnitt 6.5, beschrieben. Kurz zusammengefasst kann festgehalten werden, dass eine Klasse, die dieses Entwurfsmuster verwendet, mehrere Zustände annehmen kann, die das Verhalten dieser Klasse bzw. einer Instanz dieser Klasse beinflussen. Im Falle eines Grafikprogramms und des Topologieeditors werden die einzelnen Zustände als Werkzeug bezeichnet. Das Objekt, das diese Werkzeuge verwendet, stellt die Grafikoberfläche, in diesem Fall eine Ansicht der Topologie, dar. Diese Oberfläche fängt Mausbewegungen und -eingaben ab und leitet diese an das aktuelle Werkzeug weiter. Dieses Werkzeug entscheidet schließlich, welche Änderungen an der Oberfläche vorgenommen werden. Abbildung 7.1 zeigt die Struktur des Editors.

Der Topologieeditor registriert sich als MouseListener und MouseMotionListener. Die Events, die bei ihm eingehen, leitet er an seinen aktuellen Zustand bzw. sein aktuelles Werkzeug weiter. Diese Weiterleitung geschieht beispielsweise folgendermaßen:

```
public void mouseClicked(MouseEvent e) {
  tool.mouseClicked(e);
}
```

Die Anzahl der Werkzeuge ist im Falle des Topologieeditors noch überschaubar:

- MoveTool : wird zum Bewegen der Agenten auf der Oberfläche verwendet. Folglich überwacht dieses Werkzeug, ob und an welcher Stelle die linke

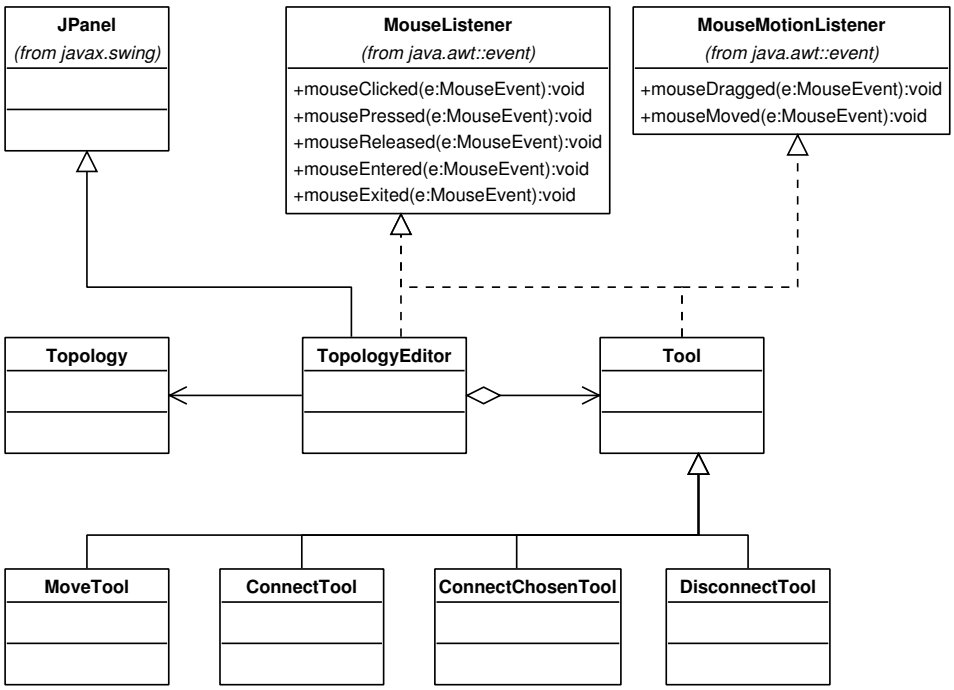


Abbildung 7.1: Aufbau des Topologieeditors

Maustaste zum Auswählen eines Agenten betätigt wurde, wohin die Maus sich bewegt und wo die Maustaste wieder losgelassen wird.

- ConnectTool : wird zur Erzeugung von Verbindungen zwischen jeweils zwei Agenten erzeugt. Hierbei überwacht das Werkzeug die Mausklicks und die Positionen, an denen diese erfolgen. Hierbei wird zwischen einem ersten und einem zweiten Mausklick unterschieden, wobei jeder Klick auf einen Agenten gemacht werden muss.

- ConnectChosenTool : ähnelt dem vorangegangenen Werkzeug. Die Arbeitsweise ist die gleiche. Der Unterschied besteht darin, dass, bevor das Werkzeug eingesetzt werden kann, eine Klasse ausgewählt wird, die zum Verbinden der Agenten verwendet wird.

- DisconnectTool : dieses Werkzeug überwacht Mausklicks und dessen Position. Erfolgt ein solcher in unmittelbarer Nähe oder auf einer Verbindung, wird diese gelöscht.

Die genaue Beschreibung der Verwendung des Topologieeditors befindet sich in Abschnitt 5.3.3.

7.2 Visualisierung und Animation

Die Visualisierung ist eine der wichtigsten Komponenten von DisASTer. Sie soll dem Anwender einen schnellen und leicht verständlichen Überblick über das Verhalten verteilter Algorithmen geben. Dieser Abschnitt beschäftigt sich mit der Implementierung und den verwendeten Techniken. Der Entwurf der Visualisierung wird in Abschnitt 6.4 beschrieben.

Die grafische Darstellung erfolgt über die *2D Graphics*-API, speziell über die Klasse `java.awt.Graphics2D`. Diese Klasse erweitert die `Graphics`-Klasse, die von den einzelnen Swing-Komponenten zum Zeichnen verwendet wird. Mittlerweile werden nur noch `Graphics2D`-Objekte zum Zeichnen der einzelnen Komponenten verwendet. Allerdings hat sich die Schreibweise der Methoden nicht verändert, was den Anschein erweckt, es würde nur mit einfachen `Graphics`-Objekten gearbeitet. Um die erweiterte Funktionalität der *2D Graphics*-API nutzen zu können, müssen also alle `Graphics`-Objekte nach `Graphics2D` gecastet werden:

```
protected void paintComponent(Graphics g){
  Graphics2D g2 = (Graphics2D) g;
}
```

Grund für diesen Schritt ist der hohe Funktionsumfang der *2D Graphics*-API, der für eine ordentliche grafische Darstellung benötigt wird. Hierbei ist vor allem das Antialiasing hervorzuheben, das zur Abrundung von Linien verwendet wird. Ohne diese Technik würden Linien, wie sie von den Pfeilen der Ablaufansicht oder den Verbindungen der Topologieansicht verwendet werden, häufig eckig erscheinen, was in Abbildung 7.2 gezeigt wird.

Abbildung 7.2: Vergleich: Kein Antialiasing / Antialiasing

Diese sauber gezeichneten Linien können mit der *2d Graphics*-API noch weiter bearbeitet werden, was bei den Einstellungen der Verbindungen der Topologieansicht zum Einsatz kommt. Es besteht die Möglichkeit, die Linien gestrichelt, gepunktet, dicker oder dünner zu zeichnen. Diese Anpassungen erfolgen über `Strokes`-Objekte, die den Verlauf von Linien beschreiben. Im folgenden Code wird ein `Stroke`-Object erzeugt, mit dem Linien gestrichelt dargestellt werden:

```
BasicStroke dashed = new BasicStroke(1.0f,
  BasicStroke.CAP_BUTT,
  BasicStroke.JOIN_ROUND,
```

```
        10.0f, new float[] {8.0f, 5.0f}, 0.0f);
```

Das dort angegebenene float-Feld (`new float[] {8.0f, 5.0f}`) gibt die einzelnen Segmentabstände an, die im Laufe einer Linie wiederholt werden. Dabei wird zwischen sichtbarem Segment und Zwischenraum unterschieden. Im obigen Beispiel wird also immer ein Teilstück der Länge 8.0 Pixel gezeichnet, gefolgt von einem Zwischenraum von 5.0 Pixel. Dieses Schema erstreckt sich über die gesamte Länge einer Linie. Es können auch Felder mit mehr als zwei Segmenten angegeben werden, um z.B. eine Punkt-leer-Strich-Abfolge zu erzeugen.

Eine weitere Funktionalität, die in den Visualisierungen von DisASTer eine Verwendung findet, ist der Farbverlauf (Gradient). Ein Farbverlauf kann alternativ zu einer einfachen Füllfarbe verwendet werden. Der folgende Code zeigt, wie die aktuelle Füllfarbe auf einen Farbverlauf gesetzt wird:

```
        g2.setPaint(new GradientPaint(x1, y1, Color.BLUE,
          x2, y2, Color.RED));
```

Die Erzeugung eines solchen Verlaufs kann man sich einfach vorstellen. Auf einem Tablett werden an zwei verschiedenen Punkten zwei Farbeimer mit unterschiedlichen Farben ausgeleert. Die Farben verteilen sich gleichmäßig über das ganze Tablett, bis sie zusammenstoßen. An dieser Stelle vermischen sie sich und bilden den eigentlichen Verlauf. Der Konstruktor der Klasse `GradientPaint`, der im obigen Beispiel zu sehen ist, verdeutlicht dieses Vorgehen. Es werden zwei Punkte (`x1`,`y1` und `x2`,`y2`) angegeben, auf die jeweils eine andere Farbe „gegossen“ wird. In der Visualisierung werden Farbverläufe vor allem für die Darstellung der Agenten verwendet. Durch einen Farbverlauf sieht es so aus, als besäßen die Agenten Schattierungen, wodurch ein einfacher aber optisch ansprechender 3D-Effekt erzielt wird.

Ein großer Vorteil eines `Graphics2D`-Objekts ist, dass es sich problemlos skalieren lässt. Diese Technik wird zum Zoomen der Ansichten verwendet. Mit einem einfachen `Graphics`-Objekt ist eine Skalierung nur mit dem Einsatz eines `BufferedImage` möglich. Dabei hätte dann die Ansicht in ein solches Bild portiert werden müssen, um dieses dann zu vergrößern oder zu verkleinern. Dieser Weg würde allerdings viel Rechenleistung in Anspruch nehmen, wohingegen die Skalierung des `Graphics2D`-Objekts wesentlich performanter ist.

Um die Visualisierung ein wenig lebendiger zu gestalten, werden Animationen verwendet. Diese werden vom Controller (Abschnitt 6.5) mit Hilfe von Threads realisiert. Dabei wird ein neuer Thread gestartet, der im Laufe eines Zeitschritts die Ansichten in regelmäßigen Abständen benachrichtigt, sich neu zu zeichnen. Der prozentuale Fortschritt eines Zeitschrittes wird mit der `double`-Variablen `progress` an die Ansichten und deren Renderer weitergegeben. Eine genaue Beschreibung der Erzeugung von Animationen mit Hilfe dieser Variablen findet sich in Abschnitt 5.2.4.

7.3 Klassenladen

DisASTer ist hochgradig erweiterbar. Der Anwender kann an sehr vielen Stellen eigene Klassen verwenden. Hierunter fallen z.B. Erweiterungen der Klasse `Agent`, `Connection`, `TopologyManager`, `View` oder `Renderer`. Um seine selbst geschriebenen Klassen in das Programm zu integrieren, wird der Benutzer mit einem Dateiauswahldialog der Klasse `JFileChooser`, aus der Swing-Bibliothek, aufgefordert, diese aus dem Dateisystem auszuwählen. Um das Auswählen anderer Dateitypen zu vermeiden, verwendet dieser Dialog einen sogenannten `FileFilter`, mit dem es möglich ist, nur bestimmte Dateitypen - in diesem Fall `.class`-Dateien - anzuzeigen.

Von der Auswahl der Klassendatei aus dem Dateisytem, bis hin zu einem Java-Objekt dieser Klasse, ist es allerdings noch ein weiter Weg. Zunächst muss die Klasse in die Virtuelle Maschine geladen werden. Diese Aufgabe übernehmen die Klassenlader (`ClassLoader`). Die gewählte Klasse liegt zunächst nur als `File`-Objekt vor, aus dem noch kein Objekt erzeugt werden kann und mit dem auch kein Klassenlader etwas anfangen kann. Es besteht jedoch die Möglichkeit, den Pfad der Datei, mit Hilfe der Methode `toURL()` der Klasse `File`, in eine URL umzuwandeln. Ein `URLClassLoader` (`java.net.URLClassLoader`) kann - was sein Name schon verät - diese URLs benutzen, um eine Klasse zu laden. Beim Erzeugen einer neuen Instanz dieses Klassenladers wird diesem ein Feld von URLs im Konstruktur übergeben. Aus diesem Feld können dann zusätzliche Klassen in das System geladen werden.

Diese Arbeitsweise funktioniert allerdings nur, solange die Klassen in keinem Package liegen. Das `File`-Objekt weiss nicht, dass es eine Javaklasse repräsentiert und es weiss auch nicht, welchem Package diese Klasse zugeordnet wird. Ein `URLClassLoader` braucht diese Informationen. Folglich muss eine URL zu einer Klasse, die in einem Package liegt, wie folgt aussehen:

```
file://pfad/zur/package.JavaKlasse.class
```

Die von dem File-Objekt generierte URL sieht allerdings so aus:

```
file://pfad/zur/package/JavaKlasse.class
```

Der Unterschied dürfte beim ersten Hinsehen vielleicht nicht auffallen. Im ersten Fall wird das Package und die Javaklasse durch einen „ . “ getrennt, wie es auch vom `Java`-Kommando bekannt ist und benötigt wird. Im zweiten Fall, welcher zur Zeit noch vorliegt, werden die beiden Elemente durch einen „/“ getrennt. Der Versuch, die gewählte Javaklasse mit der zweiten URL zu laden würde eine `NoClassDefFoundError`-Ausnahme werfen. Dieser Fehler kann aber genutzt werden, um das vorliegende Problem zu beseitigen. Tritt ein solcher Fehler auf,

wird die URL so geändert, dass der letzte „/“ mit einem „ . “ ausgetauscht wird. Anschließend wird erneut versucht, die Klasse zu laden. Der Vorgang wird so oft wiederholt, bis das Laden erfolgreich war oder bis der letzte (erste) „/“ ersetzt wurde. Dieser Algorithmus erlaubt das Laden von Klassen aus einer beliebig tief geschachtelten Package-Struktur. Ergebnis dieses Vorgangs ist ein Objekt der Klasse `Class`. Das weitere Vorgehen mit diesem Objekt wird im nächsten Abschnitt beschrieben.

7.4 Reflection und Pseudopolymorphismus

Der vorangegangene Abschnitt beschreibt die Problematik, noch nicht bekannte Klassen aus dem Dateisystem zu laden. Dieser Abschnitt geht noch einen Schritt weiter und beschäftigt sich mit der Problematik, diese neu geladenen Klassen mit Hilfe der *Reflection*-API zu bearbeiten. Die *Reflection*-API gibt Aufschluss über Methoden, Felder oder Konstruktoren von einer Klasse. Diese werden durch Java-Objekte repräsentiert und können über diese abgefragt oder aufgerufen werden. Die wichtigste Klasse in diesem Zusammenhang ist die Klasse `Class`. Ein Objekt dieser Klasse enthält Informationen über Methoden, Konstruktoren oder Attribute einer Klasse, aber auch die Klassenhierarchie, die Klasse, aus der sie abgeleitet ist, oder implementierte Interfaces. Diese Informationen werden bereits zum Klassenladen verwendet. Oft müssen die ausgewählten Klassen eine bestimmte andere erweitern. Erhält man über die im vorigen Abschnitt beschriebene Methode eine Klasse, wird überprüft welche sie erweitert bzw. welche Interfaces sie implementiert, um sicherzugehen, dass Objekte dieser Klasse zur weiteren Verarbeitung verwendet werden können. Wird die ausgewählte Klasse diesen Ansprüchen gerecht, kann nun ein Objekt dieses Typs erzeugt werden. Um dies zu erreichen, können entweder Aufrufe der `Contructor`-Objekte der Klasse oder die Methode `newInstance()` verwendet werden. Die Methode `newInstance()` setzt allerdings einen leeren Konstruktor voraus, da über sie noch keine Parameterübergabe möglich ist. Ein Konstruktoraufruf mit Parametern erfolgt über die `Constructor`-Objekte.

Die Erzeugung neuer Objektinstanzen über die *Reflection*-API findet in DisASTer sehr häufig ihre Anwendung. Oberflächlich gesehen scheint sich dieser Vorgang zunächst nur auf das Laden systemfremder Klassen zu beschränken, jedoch geht die Anwendug weit darüber hinaus. Jede Nachricht, die ein Agent versendet, wird zunächst mit Hilfe der *Reflection*-API „kopiert“, indem eine neue Instanz der Klasse dieser Nachricht erzeugt wird, und die Attribute übernommen werden. Für dieses Vorgehen sprechen zwei wichtige Gründe:

1. Broadcast. Ein Agent besitzt die Möglichkeit, eine Nachricht an alle seine erreichbaren Nachbarn zu senden. Diese Art der Kommunikation kann man als simulierten Broadcast oder besser noch als simulierten Multicast bezeichnen. Dabei muss er nur eine einzige Nachricht angeben, trotzdem erhält jeder andere Agent, der mit ihm verbunden ist, diese Nachricht. Genauer gesagt bekommt keiner wirklich die ursprünglich gesendetet Nachricht, sondern jeder bekommt eine Kopie. Damit ist sichergestellt, dass nicht jeder Agent gleichzeitig auf dasselbe Objekt zugreift. Eine Alternative zu dieser Methode, wäre die Überschreibung der `clone()`-Methode durch den Anwender. Um ihm diese Arbeit aber zu ersparen, wird die *Reflection*-API verwendet.

2. Weiterleitung. Erhält ein Agent eine Nachricht, könnte es sein, dass er diese unmittelbar wieder weiterleiten möchte. Würde an dieser Stelle dieselbe Nachricht und nicht eine Kopie versendet, würde das das ganze System durcheinander bringen, denn die Nachricht befindet sich bereits in der Liste der gesendeten Nachrichten. Bei einer Weiterleitung würde sich die gesamte Situation dieser Nachricht, Sender, Empfänger, Sendezeit usw., ändern, obwohl sie eigentlich unveränderbar in der Liste der bereits gesendeten Nachrichten liegt. Aus diesem Grund wird jede Nachricht, die von einem Agenten gesendet werden soll, zunächst kopiert und dann erst gesendet.

Eine Besonderheit der Agenten ist der Pseudopolymorphismus, der bei den `onReception()`-Methoden zum Tragen kommt. Polymorphismus bedeutet Vielseitigkeit und bewirkt die Zuweisung unterschiedlicher Klassen zu einem Objekt. So kann z.B. ein Objekt des Typs Fahrzeug den Wert eines Autos oder den eines Fahrrads annehmen. Methoden legen mit ihren Parametern ein ähnliches Verhalten an den Tag. Ursprünglich wurde damit gerechnet, dass der Polymorphismus bei den unterschiedlichen Nachrichtentypen sofort greift, der Benutzer für jeden dieser Typen eine eigene `onReception(<NachrichtenTyp>)`-Methode definieren kann und diese automatisch aufgerufen werden. Dadurch, dass die Nachrichten bei der Auslieferung zunächst von der MessageQueue, die diese Nachrichten ausliefert, auf den einfachen Typ Message gecastet werden, werden die Unterklassen dieses Typs nicht erkannt. Aus diesem Grund wurde der Pseudopolymorphismus mit Hilfe der *Reflection*-API erzeugt. Bevor eine Nachricht an einen Agenten ausgeliefert wird, überprüft die MessageQueue, ob der Agent über eine Methode verfügt, die genau diesen Typ abfängt. Ist dies nicht der Fall, wird eine Methode gesucht, die die Oberklasse dieses Typen abfängt, bis eine passende `onReception()`-Methode gefunden wurde oder bis die Oberklasse `Message` erreicht ist, für die durch die Klasse `Agent` bereits eine Empfangsmethode existiert. Dabei wird die Klasse des Nachrichtentyps mit den Parametern der Methoden verglichen, die „onReception“ heißen. Wird eine passende Methode gefunden, wird

das entsprechende `Method`-Objekt mit der `invoke(Object, Object[])`-Methode aufgerufen. Der erste Parameter dieser Methode beschreibt das Objekt, auf dem diese Methode ausgeführt wird. Der zweite Parameter, das Objekt-Feld, stellt die zu übergebenen Parameter dar. Auf diese Weise kann der Benutzer für jeden Nachrichtentyp den er verwendet, eine neue `onReception()`-Methode definieren und sicher sein, dass diese auch aufgerufen wird, sobald eine Nachricht dieses Typs ankommt.

7.5 Oberflächengestaltung

Karsten Lentzsch ist der Meinung, dass es wirklich einfach ist, schlechte Oberflächen mit Swing zu bauen (vgl. [Lentzsch03], S. 31), womit er den Nagel auf den Kopf getroffen hat.

> „Hässlich, langsam und verunstaltet - so sind viele Swing-Programme. Aber dagegen kann man was tun, und zwar schnell."

so Lentzsch weiter ([Lentzsch03], S. 31). Dass Swing als hässlich erachtet wird, liegt wohl hauptsächlich am Look-and-Feel. Zur Darstellung der einzelnen Oberflächenelemente verwendet Swing die *plugable Look-and-Feels*. Diese lassen sich zu Begin einer Anwendung bestimmen oder sogar während dem Verlauf ändern. Standardmäßig ist das *Metal*-Look-and-Feel eingestellt, was mit beissenden Farben und einer unfreundlichen Oberfläche ausgestattet ist. Dass Swing langsam ist, liegt daran, dass es im Gegensatz zu AWT oder SWT plattformunabhängig ist und nicht auf native Elemente zurückgreift. Allerdings rückt dieser Angriffspunkt bei der Rechenleistung heutiger Computer immer weiter in den Hintergrund und stellt nicht mehr wirklich ein Hindernis für den Einsatz von Swing dar. Ein weiterer Knackpunkt ist die komplizierte Erstellung von grafischen Benutzeroberflächen mit Swing. Die meisten GUI-Designer helfen dabei nur bedingt. Hier treten sehr oft dann Probleme auf, wenn der Entwickler die generierten Oberflächen manuell ändern möchte, da diese Designer mit eigenen Konfigurationen arbeiten und sobald diese geändert werden, kann der nützliche Helfer nichts mehr mit der vorliegenden Oberfläche anfangen. Eine Oberfläche in Swing per Hand zu erstellen, erfordert hingegen ein gewisses Maß an Erfahrung, was zunächst einige Zeit in Anspruch nimmt. Ist diese Hürde erst einmal genommen, steht der Entwicklung ergonomischer Oberflächen nichts mehr im Weg. Es bleibt nur noch das Problem mit dem Look-and-Feel. Aber auch hier finden sich immer mehr Lösungen. Die Anzahl neu entwickelter Look-and-Feesls steigt stetig. Angefangen hat

diese Entwicklung mit dem *Kunststoff*-Look-and-Feel von Incors[6]. Daneben finden sich mittlerweile diverse Simulationen verschiedenster Betriebssysteme oder Windowmanager, sowie Lösungen, die sich auf Geschwindigkeit oder Einfachheit spezialisiert haben. Unter dieser Auswahl unterschiedlichster Look-and-Feels findet sich mittlerweile ein passendes für jedes Anwendungsgebiet.

In DisASTer wurde das Ziel verfolgt, eine schnelle aber auch eine optisch ansprechende Oberfläche anzubieten. Hierfür wurde zunächst das bereits erwähnte *Kunststoff*-Look-and-Feel in Betracht gezogen, doch dessen Einsatz hat die Leistung des Programms, gerade im Visualisierungsbereich, deutlich verringert. Aus diesem Grund wurde ein Look-and-Feel gesucht, welches weniger Ressourcen benötigt. Mit dieser Zielsetzung arbeitet das *JGoodies*-Projekt[7] von Karsten Lentzsch. In diesem finden sich eine Reihe von Anwendungen und Look-and-Feels, die schnell und leistungsorientiert arbeiten. Eines der dort zu findenden Look-and-Feels ist das *PlasticXP*-Look-and-Feel, was auch für diese Anwendung verwendet wurde. Dieses simuliert zum Teil das Aussehen der Windows XP Oberfläche, bietet aber zusätzlich eigene Elemente, wie z.B. ein Farbverlauf in Menü- und Toolbars, um einen einfachen 3D-Effekt zu erzielen. Zusätzlich enthält es 18 Schrift- und Farbschemas, die auf den unterschiedlichsten Plattformen eingesetzt werden können, um dort ein möglichst einheitliches Erscheinungsbild zu garantieren.

Ganz ohne Anpassungen ließ sich das Look-and-Feel aber doch nicht in DisASTer einbinden. Das größte Problem stellten hier die Reiter einer `JTabbedPane` dar. Diese wurden für das *PlasticXP*-Look-and-Feel so angepasst, dass sie alle dieselbe Farbe verwenden und sich diese Farbe auch nicht mit üblichen Befehlen ändern läßt. Das Ändern der Hintergrundfarbe dieser Reiter wird aber für die Ausgabeströme (siehe Abschnitt 5.3.2, Seite 59) verwendet.

Ein weiterer Aspekt bei grafischen Oberflächen sind die verwendeten Symbole (Icons). Diese sollten gut erkennbar sein und sich farblich weitgehend mit dem Aussehen der restlichen Anwendung decken. Sun stellt eine Sammlung der gängigsten Symbole bereit[8], die sich ausgezeichnet in Swing-Oberflächen integrieren lassen. Für DisASTer wurden hauptsächliche diese und einige selbst erzeugte verwendet. In der Online-Hilfe finden sich einige Symbole der Eclipse-Symbolsammlung[9].

Neben dem Aussehen steuert der Aufbau einer grafischen Oberfläche einen wesentlichen Teil zu einer guten Benutzbarkeit bei. Aus diesem Grund finden sich in DisASTer aus anderen Anwendungen bekannte Menüstrukturen und Elemente wieder, wodurch eine schnelle und intuitive Benutzung gewährleistet werden soll.

[6] http://www.incors.org/
[7] http://www.jgoodies.com/
[8] http://developer.java.sun.com/developer/techDocs/hi/repository/
[9] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.jdt.debug.ui/icons/

7.6 Speichern von Umgebungszuständen

Der Verlauf der Simulation (siehe Kapitel 6, Seite 84) ist eine große Navigationshilfe für den Benutzer. Mit ihm hat der er die Möglichkeit, die Simulation zurück zu spulen, um den Algorithmus entweder erneut oder unter veränderten Bedingungen ablaufen zu lassen. Um dies zu gewährleisten, speichert der Verlauf die einzelnen Zustände der Umgebung ab. Dies geschieht über Serialisierung eines Objekts der Klasse `EnvironmentState`. Dieses serialisierte Objekt wird allerdings zunächst nicht wie üblich in eine Datei geschrieben oder über ein Netzwerk geleitet, sondern es soll in die Liste des Verlaufs aufgenommen werden. Ein solches serialisiertes Objekt zu erhalten, ist zunächst nicht ganz einfach, denn es wird in einen Strom (`ObjectOutputStream`) geschrieben. Um nun dieses Objekt aus diesem Strom wieder herauszulesen, ohne den Umweg über eine Datei zu machen, werden die Klassen `PipedInputStream` und `PipedOutputStream` verwendet. Objekte dieser beiden Klassen werden miteinander verbunden, wodurch es möglich ist, den Inhalt des Ausgabestroms sofort wieder einzulesen. Ein kleines Problem dabei ist, dass hier eine Verklemmungsgefahr besteht, wenn sich Lesen und Schreiben gegenseitig behindern. Aus diesem Grund wird die Leseoperation in einem separaten Thread ausgeführt.

Das Verfahren, aktuelle Zustände der Umgebung zu speichern und einzulesen, wird im Programm zusätzlich bei den Einstellung verwendet. Sobald die vorgenommen Einstellungen mit „OK“ bestätigt werden, wird eine Kopie, der dort erstellten Umgebung auf die tatsächliche Umgebung übertragen. Um einen schnellen und einfachen Zugriff auf diese Kopien zu erhalten, wurde das Entwurfsmuster „Fabrikmethode“ gewählt. Hierbei stellt die sogenannte Fabrik eine statische Methode bereit, die ein solches Objekt zurückliefert. Diese Methode wird als Fabrikmethode bezeichnet. Dadurch bleibt den Objekten, die auf diese serialisierten Zustände der Umgebung zurückgreifen, erspart, diesen umständlichen Vorgang mit den Strömen selbst durchzuführen (vgl. [Gamma et. al. 96], S. 115ff).

Letzendlich kann der Zustand der Umgebung in einer Datei abgespeichert werden. Das ermöglicht den schnellen Aufruf eines Algorithmus, ohne ihn jedesmal neu zu konfigurieren. Im Falle der Auslagerung der Umgebung in eine Datei wird zusätzlich der Verlauf gespeichert. Im Falle des Verlaufs und der Einstellungen ist dies nicht nötig und würde unnötigen Speicher verbrauchen.

7.7 Verteilung mit Ant

Das Programm wird in zwei unterschiedlichen Versionen ausgeliefert (siehe Abschnitt 5.1.1). Um beide Versionen mit den nötigen Dateien auszustatten, wurde Ant[10] verwendet. Bei Ant handelt es sich um ein Java-basiertes build-Werkzeug. Es ähnelt ein wenig dem aus der Unix/Linux-Welt bekannten Make. Die Entwickler von Ant wollten mit ihrem Projekt vor allem die Schwierigkeiten beheben, die Make mit sich bringt. Neben einer plattformunabhängigen Lösung wollten sie vor allem eine einfache Syntax anbieten, die bei Make nicht gegeben ist. Beides lässt sich durch das „Dreamteam“ Java und XML lösen: Java, als plattformunabhängige Programmiersprache, und XML, als plattformübergreifendes Dateiformat.

Die Entscheidung Ant zu verwenden, lässt sich durch zwei Gründen belegen: zum einen wurde DisASTer unter Eclipse[11] entwickelt, das eine sehr gute Integration von Ant besitzt, und zum anderen hat sich Ant in der Java-Welt mittlerweile als Quasistandard durchgesetzt und wird von vielen Entwicklern eingesetzt.

Ant arbeitet mit den sogenannten build-Dateien, die im XML-Format vorliegen. In einer solchen build-Datei werden Befehle aufgelistet, die das Programm dann beim Aufruf ausführt. Ein Befehl wird in Ant auch als Task bezeichnet. Gerade für die Javaentwicklung stellt Ant eine Reihe nützlicher Tasks zur Verfügung. So gibt es beispielsweise Tasks zum Kompilieren (javac), zum Ausführen (java), zum Archivieren (jar) oder zum Erzeugen von JavaDoc. Daneben gibt es Tasks, mit denen ein Zugriff auf das Dateisystem möglich ist. Dazu gehören Kopieren, Verschieben, Löschen und viele weitere nützliche Befehle. Zusätzlich können eigene Tasks definiert werden, wodurch Ant in vielen Bereichen eingesetzt werden kann. Diese Tasks müssen einer Gruppe (target) zugeordnet sein. Eine solche Gruppe beschreibt meist eine bestimmte Aktion. Es können Abhängigkeiten zwischen den einzelnen Gruppen erzeugt werden, z.B. kann eine Befehlsgruppe, die das Erzeugen einer Jar-Datei bewirken soll, erst ausgeführt werden, wenn die zu archivierenden Klassen erzeugt wurden. Die Befehle, die zum Erzeugen dieser Klassen zuständig sind, befinden sich in einer anderen Gruppe. Ein Auszug einer verwendeten build-Datei wird in Listing 7.1 dargestellt.

```
1 <project name="DisASTer" default="jar">
2
3   <property name="src" location="src"/>
4   <property name="bin" location="bin"/>
5   <property name="doc" location="doc"/>
6   <property name="jarfile" location="lib/disaster.jar"/>
7   <property name="manifest" location="MANIFEST.MF"/>
8
9   <target name="compile">
10    <javac srcdir="${src}" destdir="${bin}" />
11  </target>
```

[10] http://ant.apache.org/
[11] http://www.eclipse.org/

```
12
13   <target name="javadoc" depends="compile">
14     <javadoc destdir="${doc}/api" access="protected"
15         sourcepath="${src}" classpath="${bin}">
16         <link href="http://java.sun.com/j2se/1.4.1/docs/api"/>
17     </javadoc>
18   </target>
19
20   <target name="jar" depends="compile, javadoc">
21     <jar destfile="${jarfile}" basedir="${bin}"
22         manifest="${manifest}" update="true">
23       </jar>
24   </target>
25 </project>
```

Listing 7.1: Ant build-Datei

Um nun ein solches Antskript ausführen zu lassen, genügt es, in das Verzeichnis zu wechseln, in dem sich dieses Skript befindet und den `ant`-Befehl aufzurufen. Besitzt die Datei den Namen build.xml, wird sie automatisch erkannt und ausgeführt. Andernfalls kann dem Befehl die Datei mit dem zu erzeugenden Skript übergegeben werden.

Die beiden unterschiedlichen Versionen von DisASTer lassen sich mit Hilfe eines einzigen Antskripts generieren. Dieses erzeugt zwei Zip-Archive, in denen sich die unterschiedlichen Versionen befinden. Zusätzlich befindet sich in jeder dieser Versionen eine build-Datei, um das Programm erneut erzeugen zu lassen. Um also aus der wesentlich kleineren Quellcode-Version die vollständige Version zu erhalten, genügt es, das beiliegende Antskript auszuführen.

7.8 Hilfesystem

Das Kapitel 5, über die Verwendung von DisASTer, wird zusammen mit der API-Dokumentation (JavaDoc) als Online-Hilfe bereitgestellt. Dadurch sind alle wichtigen Informationen über und für das Programm aus diesem abrufbar. Dadurch muss kein zusätzliches Programm gestartet werden, in dem die externe Hilfe zu lesen ist, sei es ein Webbrowser oder beispielsweise der Acrobat Reader.

Um ein solches Hilfesystem in Java zu realisieren, wird das von Sun selbst entwickelte JavaHelp[12] in der Version 2 verwendet. JavaHelp stellt eine Bibliothek dar, die bereits viele Elemente beinhaltet, um Online-Hilfen in der Form zu erstellen, wie sie aus anderen Programmen bekannt sind. Zu diesen Elementen zählt eine Übersicht in Form eines Inhaltsverzeichnisses, eine Suchfunktion und ein Index über die wichtigsten Begriffe rund um ein Programm. Die Hilfetexte

[12] http://java.sun.com/products/javahelp/

selbst können dann einfach in HTML verfasst werden. Zur Ansicht dieser Texte wird eine auf Java basierte Browserkomponente benutzt.

Um besser durch die einzelnen Seiten zu navigieren, wird ein Inhaltsverzeichnis angezeigt. Darüber hinaus bietet das System die Möglichkeit, eine Volltextsuche für den gesamten Inhalt der Hilfe erzeugen zu lassen. Zusätzlich zu diesen beiden Navigationshilfen kann noch ein Index erstellt werden, über den der Benutzer die wichtigsten Begriffe in alphabetischer Reihenfolge aufgelistet bekommt.

Im Folgenden werden die Schritte erläutert, die für das Erzeugen einer Online-Hilfe mit Javahelp, speziell der von DisASTer, notwendig sind. Zunächst gilt es einmal, den Inhalt zu erzeugen. Hierzu gehört die JavaDoc-Dokumentation, die ja bereits in HTML vorliegt und der Text aus Kapitel 5. Um diesen in JavaHelp verwenden zu können, muss er in das HTML-Format konvertiert werden. Bisher liegt er im TeX-Format vor. Die Umformatierung erfolgt aber nicht von Hand. An dieser Stelle kommt das Programm latex2html zum Einsatz. Hierbei handelt es sich um ein Perl-Skript, das - wie der Name schon sagt - LaTeX-Dokumente in das benötigte HTML-Format konvertiert. Für diese Aufgabe wurden auch andere Programme wie z.B. TtH (TeX to HTML) getestet. Allerdings überwiegten die Vorteile des Perl-Skripts. Der größte Vorteil von latex2html gegenüber TtH besteht darin, dass für jedes Kapitel, Unterkapitel und jeden Abschnitt eine separaten Seite erzeugt wird. TtH stopft das ganze Kapitel auf eine HTML-Seite. Aber durch die Aufteilung auf mehrere Seiten wird das Hilfesystem viel übersichtlicher, da auch wirklich nur das Kapitel angezeigt wird, das der Benutzer eigentlich sehen will. Ins nächste Kapitel zu blättern stellt dann immer noch kein Problem dar, denn latex2html setzt automatisch Verweise auf das vorangegangene, nächste und übergeordnete Kapitel. Ganz davon abgesehen, dass dem Benutzer später im fertigen Hilfesystem das Inhaltsverzeichnis zur Verfügung steht.

Da jetzt der Inhalt für das Hilfesystem vorliegt, kann detailierter auf die Arbeitsweise von JavaHelp eingegangen werden, um zu sehen, wie es mit diesen Webseiten arbeitet und wie diese eingebunden werden. Das alles wird über XML-Konfigurationsdateien gelöst. Die wichtigste Datei hierbei ist das HelpSet. Ein HelpSet wird durch eine XML-Datei repräsentiert, die die Endung „.hs“ besitzt. Hier werden alle weiteren XML-Konfigurationsdateien und Einstellungen der Ansichten (Views) angegeben. Diese Views sind unter anderem für den Aufbau des Inhaltsverzeichnisses, der Volltextsuche und des Index verantwortlich. Die folgenden Abschnitte erklären Schritt für Schritt die Erzeugung des HelpSets, ausgehend von dessen Rumpf (Listing 7.2):

```
1  <?xml version='1.0' encoding='ISO-8859-1' ?>
2  <!DOCTYPE helpset
3      PUBLIC
4      "-//Sun Microsystems Inc.
5      //DTD JavaHelp HelpSet Version 1.0//EN"
6      "http://java.sun.com/products/javahelp/helpset_1_0.dtd">
7    <helpset version="1.0">
8      <title>DisASTer Help</title>
9      <!-- ... -->
10   </helpset>
```

Listing 7.2: Helpset-Rumpf

Die zweitwichtigste Datei eines HelpSets ist die Mapping-Datei (Endung .jhm), in der alle verwendeten HTML-Seiten, aber auch Grafiken, einer eindeutigen Bezeichnung zugeordnet (gemapt) werden. Ein Auszug der verwendeten Mapping-Datei, in Listing 7.3, zeigt die Arbeitsweise:

```
1  <map version="1.0">
2    <!-- Mapping einer Grafik -->
3    <mapID target="tocopen" url="images/toc_open.gif" />
4    <!-- Mapping einer Webseite -->
5    <mapID target="api" url="api/overview-summary.html" />
6    <!-- ... -->
7  </map>
```

Listing 7.3: Auszug der Mapping-Datei

Jedes Element, ob HTML-Seite oder Grafik, wird als `mapID` erzeugt. `target` beinhaltet die eindeutige Bezeichnung dieses Elements und `url` verbindet diese Bezeichnung mit einer Datei. Um diese Elemente in Zukunft zu verwenden, werden diese nur noch über ihren mit `target` festgelegten Bezeichner angesprochen. Hierzu aber gleich mehr, denn zuerst muss die Mapping-Datei in das HelpSet integriert werden. Hierzu wird folgender Code in die HelpSet-Datei (.hs) eingefügt (Listing 7.4):

```
1  <maps>
2    <homeID>main</homeID>
3    <mapref location="disaster.jhm"/>
4  </maps>
```

Listing 7.4: Integration der Mappingdatei

`maps` leitet die Einstellungen der Mapping-Datei ein. Unter `location` wird der Ort dieser Datei angegeben.

Damit wären alle Vorbereitungen abgeschlossen und es kann mit der Erstellung des Inhaltsverzeichnisses begonnen werden. Dieses wird ebenfalls in einer separaten Konfigurationsdatei definiert. Ein Inhaltsverzeichnis (engl. table of contents, toc) besitzt eine gewisse Hierarchie, die aus Kapitel, Unterkapitel, Abschnitte usw. besteht. Daraus ergibt sich eine Baumstruktur, mit der das Inhaltsverzeichniss in der Online-Hilfe selbst und in der entsprechenden Konfigurationsdatei (Listing 7.5) dargestellt werden kann. Ein Auszug aus der verwendeten

Konfigurationsdatei macht dies deutlich:

```
1  <toc version="1.0" categoryclosedimage="tocclosed"
2    categoryopenimage="tocopen"
3    topicimage="topic">
4      <tocitem text="Benutzung" target="node2">
5      <tocitem text="Installation" target="node3"/>
6      <!-- ... -->
7    </tocitem>
8
9  </toc>
```

Listing 7.5: Auszug Inhaltsverzeichnis

Jedes Kapitel, jedes Unterkapitel und jede Seite wird durch ein `tocitem` repräsentiert. Diese können beliebig tief ineinander geschachtelt werden, wodurch die Baumstruktur entsteht. Jedes `tocitem` besitzt einen `text`, der im Inhaltsverzeichnis angezeigt wird. Die Seite, auf die der Eintrag zeigt, wird durch `target` festgelegt. Hier wird das für die entsprechende Seite definierte Mapping verwendet. Ein weiterer Einsatz der Mappings wird im Wurzelelement `toc` deutlich. Über `categoryclosedimage`, `categoryopenimage` und `topicimage` kann jeweils das Mapping einer Grafik angeben werden, die für diesen Elementtyp (geöffnetes Verzeichnis, geschlossenes Verzeichnis oder Thema) verwendet werden soll. Den einzelnen Einträgen können zusätzlich spezielle Grafiken zugewiesen werden. Nachdem das komplette Inhaltsverzeichnis mit der oben vorgestellten Technik erstellt wurde, muss es noch dem HelpSet bekannt gemacht werden, indem diesem eine View hinzugefügt wird (Listing 7.6):

```
1  <view>
2    <name>TOC</name>
3    <label>Contents</label>
4    <type>javax.help.TOCView</type>
5    <data>DisasterTOC.xml</data>
6  </view>
```

Listing 7.6: View-Integration des Inhaltsverzeichnisses

Eine solche View verfügt über einen Namen (`name`) und ein Label (`label`), welches später im Reiter mit Inhalt, Suche und Index angezeigt wird. Zusätzlich wird in `type` die Klasse angegeben, die zur Darstellung der View verwendet werden soll. In diesem Fall wird die Klasse `javax.help.TOCView` verwendet, die Teil der Javahelp-Bibliothek ist. Das `data`-Element zeigt auf die Konfigurationsdatei, in der vorhin das Inhaltsverzeichnis definiert wurde.

Javahelp besitzt eine Volltextsuche über den gesamten Inhalt der Hilfe. Diese Suche erfolgt allerdings nicht über die HTML-Seiten, sondern über eine eigens für diesen Zweck erstellte Suchdatenbank. Diese Datenbank wird mit `jhindexer`, einem einfachen Konsolenprogramm, das der Javahelp-Bibliothek beiliegt, erzeugt. Um die Suchfunktion in dem angelegten Helpset zu verwenden, wird eine neue View erzeugt und hinzugefügt (Listing 7.7):

```
1   <view>
2     <name>Search</name>
3     <label>Search</label>
4     <type>javax.help.SearchView</type>
5     <data engine="com.sun.java.help.search.DefaultSearchEngine">
6       JavaHelpSearch
7     </data>
8   </view>
```

Listing 7.7: View-Integration der Volltextsuche

Eine weitere hilfreiche Ansicht ist der Index. Hier findet der Benutzer wichtige Stichworte, die ihn zu dem entsprechenden Kapitel führen, alphabetisch geordnet. Der Index wird, ähnlich dem Inhaltsverzeichnis, mit Hilfe einer XML-Konfigurationsdatei erzeugt. Listing 7.8 zeigt den Aufbau dieser Konfigurationsdatei.

```
1 <index version="1.0">
2   <indexitem text="Ablauf und Steuerung" target="node26" />
3
4   <!-- ... -->
5
6   <indexitem text="Views" target="node16" />
7 </index>
```

Listing 7.8: Konfiguration für den Index

Der Index wird mit dem `index`-Tag eingeleitet, jeder Eintrag wird durch ein `indexitem` erzeugt. Da das Anlegen der einzelnen Einträge mit großem Aufwand verbunden ist, wurde das Programm Helen[13] getestet. Helen dient zur Erstellung von Helpsets und soll die Erzeugung von Inhaltsverzeichnis, Index und Volltextsuche unterstützen. Da das Inhaltsverzeichnis und die Volltextsuche ohne Hilfe bereits erstellt wurden, blieb nur noch der Index. Die Hoffnung lag dabei auf einer intelligenten Untersuchung der Hilfetexte, die eine Liste an Wörtern liefert, aus denen die relevanten Wörter in den Index aufgenommen werden können. Leider blieb diese Form der Unterstützung aus. Statt dessen konnten nur die einzelnen Kapitel in den Index aufgenommen werden. Ein weiteres Produkt, Javahelp Plug-In von Light Development[14], bietet eine ähnliche Funktionalität an. Aus diesem Grund wurde zunächst auf einen ausführlichen Index verzichtet. Um den Index in das Hilfesystem zu integrieren, wird dem Helpset eine neue Ansicht hinzugefügt (siehe Listing 7.9).

[13] http://www.software7.de
[14] http://www.lightdev.com

```
1   <view>
2     <name>Index</name>
3     <label>Index</label>
4     <type>javax.help.IndexView</type>
5     <data>DisasterIndex.xml</data>
6   </view>
```

Listing 7.9: View-Integration des Index

Zur Darstellung des Index wird die Klasse `javax.help.IndexView` verwendet. Im `data`-Tag befindet sich die Referenz auf die vorher angelegte XML-Konfigurationsdatei.

Um nun das fertige Hilfesystem in ein Programm einzubinden, gibt es mehrere Möglichkeiten. Bei den Demos zu JavaHelp liegt der HelpViewer bei, mit dem es möglich ist, ein HelpSet auszuwählen und dieses zu betrachten. Beim Start kann bereits ein HelpSet übergeben werden, das dann in diesem HelpViewer angezeigt wird. Eine andere Möglichkeit wäre ein HelpBroker, dem ein HelpSet übergeben wird und der für dieses ein Fenster anzeigt. Da aber keinerlei Einfluss, geschweige denn irgendwelche Änderungen an diesem Fenster vorgenommen werden können, wird auf eine dritte Möglichkeit zurückgegriffen. Der Klasse JHelp kann ebenfalls ein HelpSet übergeben werden. Hierbei handelt es sich um eine Swing-Komponente, die in einem selbst definierten Fenster abgebildet und bei Bedarf bearbeitet werden kann. Die Komponente zeigt dann das übergebene Hilfesystem im üblichen Stil an. Um ein solches JHelp zu erzeugen, wird zunächst das HelpSet über seine URL gesucht. Mit diesem HelpSet-Objekt kann schließlich die JHelp-Komponente erzeugt werden (Listing 7.10):

```
1     ClassLoader cl = getClass().getClassLoader();
2     URL hsURL = HelpSet.findHelpSet(cl, "disaster.hs");
3     HelpSet hs = new HelpSet(null, hsURL);
4     JHelp help = new JHelp(hs);
```

Listing 7.10: Erzeugen eines JHelp-Objekts

Um die fertige Online-Hilfe anzuzeigen, muss nur noch das Fenster, in dem sich die JHelp-Komponente befindet, geöffnet werden (vgl. [Sun03]).

7.9 Test

Dieser Abschnitt beschäftigt sich mit den eingesetzten Testtechniken, die verwendet wurden, um das Programm auf Fehler zu prüfen.

7.9.1 Statische Analyse

Während und nach der Implementierungsphase wurde eine statische Analyse des Quellcodes durchgeführt. Im Normalfall sollte diese allerdings nicht vom Entwickler selber, sondern von einem projektfremden Entwickler erfolgen, um eine objektivere Betrachtung zu erhalten. Da dies im Falle einer Diplomarbeit nicht üblich ist, wurde die statische Analyse zeitversetzt zur Implementierung durchgeführt, um dadurch einen gewissen Abstand zu den erstellten Programmteilen zu bekommen, der es erlaubt, den Quelltext ein wenig objektiver zu sehen.

Dadurch, dass die Analyse häufig zu Optimierungen des bestehenden Codes führt, muss diese in mehreren Schritten erfolgen, um auch den optimierten Code erneut zu überprüfen. Ergebnisse dieser Optimierungen sind in der Regel bessere Aufteilung von Methoden oder Reduzierungen von Schleifendurchläufen.

7.9.2 Systemtest

Mit Hilfe zahlreicher Beispiele und Konfigurationen wurde das Programm auf mögliche Programmierfehler und falsche Eingaben des Anwenders getestet, aber auch auf korrektes Verhalten. Dabei wurde schon bei der Implementierung darauf geachtet, dem Benutzer die Möglichkeit zu nehmen, falsche Eingaben zu machen. So wird beispielsweise die Funktionalität des Einzelschritt-Knopfes aufgehoben, solange ein Zeitschritt noch animiert wird. Der Benutzer könnte diesen Knopf zwar weiter sehr häufig hintereinander drücken, allerdings ohne irgendeine Wirkung zu erzielen. Völlig unnütze Eingaben werden von vorne herein unterbunden. So wird beispielsweise der Startknopf deaktiviert, nachdem er gedrückt wurde, und wird erst wieder aktiviert, sobald der Anwender die Simulation über Stopp oder Pause anhält. Die Deaktivierung von Knöpfen, deren Funktionalität nicht verfügbar ist, wird sehr häufig angewendet.

Gerade die Steuerung hat einen hohen Stellenwert bei den Tests erhalten, da hier die meisten Fehler auftreten können. So wurde auch häufig der Wechsel zwischen den unterschiedlichen Steuermodi getestet, um sicher zu gehen, dass dieser auch wirklich ohne negative Nebenwirkungen von statten geht.

Ein anderer Punkt, der vor allem bei den Systemtests zum Tragen kam, waren die Ansichten. Diese sind nur sehr schwer mit anderen Testverfahren zu prüfen, da sie in der grafischen Oberfläche dargestellt werden und es sehr umständlich ist, diese zu testen. Auch hierbei wurden übertriebene Eingaben gemacht, wie das schnelle Ändern der Größe des Fensters oder ständiges Zoomen der Ansichten.

Das Laden und Speichern der Umgebung wurde ebenfalls einem Systemtest unterzogen. Wichtig war hier zum einen das korrekte Einlesen der gespeicherten Daten, aber auch das korrekte Verhalten des Programms, dessen Bedienelemente

sich an die neue Umgebung anpassen müssen. Wird z.B. eine Umgebung mit ihrem Verlauf gespeichert, muss es möglich sein, nach dem Laden dieser Umgebung in den Verlauf direkt eingreifen zu können.

Das Klassenladen wird noch einmal gesondert betrachtet. Der Benutzer bekommt an vielen Stellen die Möglichkeit, eigene Klassen in das System zu integrieren. Dabei muss das Programm korrekt entscheiden, ob es sich bei der ausgewählten Datei um eine Javaklasse handelt, ob diese Klasse überhaupt geladen werden kann - hierzu muss sie sich im Klassenpfad der Anwendung befinden - und ob die Klasse den benötigten Anforderungen entspricht und über kurz oder lang die gewünschte Klasse erweitert oder das benötigte Interface implementiert.

Allgemein ist es wichtig, dass der Benutzer bei auftretenden Fehlern eine Rückmeldung erhält, die ihm Aufschluss über die Gründe des Fehlschlagens gibt. Auch diese Fälle werden im Systemtest berücksichtigt.

7.9.3 Klassentests

Klassentests, auch Unittests genannt, werden dazu verwendet, Methoden auf korrektes Verhalten zu überprüfen. Darunter fällt z.B. die Rückgabe eines korrekten Wertes, auch unter Eingabe fehlerhafter Parameter. Um Klassentests in Java durchführen zu lassen, wird das Rahmenwerk JUnit[15] eingesetzt. Dieses ist in Opensource verfügbar und hat sich mittlerweile als gängigstes Testwerkzeug für die Javaentwicklung etabliert.

Die Klassentests, die für DisASter durchgeführt wurden, beziehen sich hauptsächlich auf das Verhalten der Kontrollinstanz, die die Änderungen des Modells beinflusst. Ein Problem, das in vielen Artikeln oder Einführungen über Unittests, speziell mit JUnit, übergangen wird, sind Methoden ohne Rückgabewert, die in der Klasse `Controller` ausschließlich vorhanden sind. Um diese Methoden auf mögliche Fehler zu überprüfen, müssen die von den Methoden erzeugten Seiteneffekte überprüft werden. Die zu überprüfenden Seiteneffekte umfassen:

Zeit / Uhr Die Umstellung der Zeit muss korrekt erfolgen. Das bedeutet, sie darf nicht stehen bleiben. Nach einem Zeitschritt muss sie um einen Wert erhöht sein. Bei einem Rückschritt muss die korrekte alte Zeit wieder hergestellt worden sein. Dabei wird zwischen dem synchronen und asynchronen Modus unterschieden.

Konsistenz Die Umgebung muss konsistent bleiben. Es dürfen z.B. keine Nachrichten verloren gehen oder doppelt auftauchen.

Kommunikation Über den Controller wird ebenfalls die Umgebung getestet.

[15] http://www.junit.org

Diese muss die Nachrichten korrekt ausliefern, die Empfangszeiten richtig setzen und die Nachrichten in die richtigen Listen eintragen.

Controller-Zustände Die Zustände des Controllers müssen korrekt arbeiten. Ist ein bestimmter Zustand gewählt, muss auch dieser agieren und kein anderer. Die Umstellung auf einen anderen Controller-Zustand muss ebenfalls auf Fehler getestet werden.

8 Beispiele

In diesem Kapitel werden einige Beispiele der beschriebenen Algorithmen aus Kapitel 3 vorgestellt. Hierbei wird vor allem Wert auf die Implementierung gelegt, um zu zeigen, wie diese Algorithmen für DisASTer umgesetzt werden können. Dazu wird in der Regel eine Klasse pro Algorithmus vorgestellt, in der das Verhalten eines Prozesses in dem betreffenden Algorithmus erzeugt wird.

8.1 Logische Uhren

Das Problem der logischen Uhren wird in Abschnitt 3.1.2 beschrieben. Es geht dabei um die Beziehungen zwischen Ereignissen in unterschiedlichen Prozessen. Logische Uhren sollen dabei helfen, festzulegen, in welcher logischen Reihenfolge diese Ereignisse zueinander stehen, also welches Ereignis zeitlich vor welchen Ereignissen anzusiedeln ist.

Lamport-Methode Die logische Uhr von Lamport wird durch einen einfachen Zähler repräsentiert, der bei jedem Ereignis hochgezählt wird. Ein Ereignis könnte eine Aktion sein, die ein Prozess ausführt, darunter fallen auch das Senden und das Empfangen von Nachrichten.

Die logische Lamport Uhr lässt sich mit zwei einfachen Regeln beschreiben:

LC1 L_i wird inkrementiert, bevor ein Ereignis in Prozess p_i veranlasst wird:

$$L_i := L_i + 1$$

LC2 a. Wenn ein Prozess p_i eine Nachricht m sendet, gibt er m auch den Wert $t := L_i$ mit.

b. Beim Empfang von $< m, t >$ berechnet ein Prozess p_j den Wert $L_j := max(L_j, t)$ und wendet dann LC1 an, bevor er dem Ereignis $receive(m)$ einen Zeitstempel zuweist.

Im Folgenden wird die Implementierung eines Prozesses beschrieben, der eine Lamport-Uhr verwendet. Hierzu wird eine neue Klasse erzeugt, die die Klasse Agent erweitert. Die Lamport-Uhr wird durch ein Integer-Attribut `lamportClock` repräsentiert, das im Konstruktor mit 0 initialisiert wird.

```
1 public class LamportClockAgent extends Agent {
2   private int lamportClock;
3
```

```
4   public LamportClockAgent() {
5     lamportClock = 0;
6     description = "LamportClockAgent " + getID();
7   }
```

Dieses Beispiel beschränkt die Anzahl der möglichen Ereignisse auf zwei:

1. Das Senden einer Nachricht.

2. Das Empfangen einer Nachricht.

Durch die Initialisierung eines Agenten sendet dieser eine einfache Nachricht des Typs `Message` mit seinem momentanen Zeitstempel (**LC2** a) an seinen ersten Nachbarn (`getOther()[0]`). Vor diesem Ereignis muss er, laut **LC1**, seine Uhr inkrementieren. Zusätzlich wird das Ereignis in einer Textausgabe festgehalten.

```
8   public void init(InitMessage message) {
9     lamportClock++;
10    send(new Integer(lamportClock), getOthers()[0]);
11
12    println("sending Message to " + getOthers()[0]
13      + " at " + lamportClock);
14  }
```

Schließlich fehlt noch die Ereignisbehandlung nach dem Eintreffen einer Nachricht. Hierbei wird, nach **LC2** b, die empfangene Zeit mit der eigenen verglichen und die eigene Uhr auf das Maximum dieser beiden Werte gesetzt. Anschließend wird die Zeit laut **LC1** erhöht, bevor die Nachricht in Form einer Textausgabe weiterverarbeitet werden kann.

```
15  public void onReception(Message message) {
16    int rlc = ((Integer) message.getPayload()).intValue();
17
18    lamportClock = Math.max(lamportClock, rlc);
19    lamportClock++;
20    println("received <" + message.getSender() + ", " + rlc
21      + "> at " + lamportClock);
22  }
23 }
```

Um diesen Algorithmus vernünftig in DisASTer ablaufen zu lassen, sollten mindestens zwei dieser Agenten verwendet und miteinander verbunden werden (`CompleteTopology`). In diesem Fall würden sich auch mehrere Initialisierungen eines Agenten lohnen oder eine abwechselnde Initialisierung aller Agenten.

Eindeutige Lamport-Uhr Mit der einfachen Lamport-Methode lassen sich von einander unabhängige Ereignisse nicht unbedingt in eine Beziehung bringen, sofern diese denselben Wert aufweisen. Um die Lamport-Uhr eindeutig zu machen, wird dem Zähler die Kennung des Prozesses zugefügt. Anstatt den Zähler

immer um eins zu erhöhen, wird er in Zehnerschritten erhöht und an die Einerstelle wird die Kennung des jeweiligen Prozesses eingefügt. Voraussetzung für dieses Verfahren ist eine Höchstzahl von 10 Agenten, mit einer Kennung von 0 bis 9. Um das Verfahren für noch mehr Agenten zu verwenden, könnte die Zählung auch in Hunderterschritten usw. erfolgen.

Die Implementierung eines Agenten mit einer eindeutigen Uhr wird im Folgenden vorgestellt. Der Agent verfügt über einen Integer-Wert, der die Zeit angibt. Dieser wird im Konstruktor mit der ID des Agenten initialisiert.

```
1 public class UniqueClockAgent extends Agent {
2   private int uniqueClock;
3
4   public UniqueClockAgent() {
5     uniqueClock = getID();
6     description = "UniqueClockAgent " + getID();
7   }
```

Die Initialisierung sendet, wie im vorangegangenen Beispiel, eine Nachricht an den nächsten Agenten. Vor dem Senden dieser Nachricht, die den Zeitstempel des Agenten enhält, wird dieser erhöht.

```
8   public void init(InitMessage message) {
9     uniqueClock += 10;
10    send(new Message(getOthers()[0], new Integer(uniqueClock)));
11
12    println("sending Message to " + getOthers()[0]
13      + " at " + uniqueClock);
14  }
```

Nach dem Empfang einer solchen Nachricht wird, wie vorher auch, die Uhr auf das Maximum beider gesetzt. Dabei könnte aber die Kennung des Senders in die logische Uhr des Empfängers geraten. Um dies zu verhindern, wird die letzte Stelle entfernt und mit der ID ersetzt.

```
15  public void onReception(Message message) {
16    int rlc = ((Integer) message.getPayload()).intValue();
17
18    uniqueClock = Math.max(uniqueClock, rlc);
19    uniqueClock += 10;
20    uniqueClock -= (uniqueClock % 10);
21    uniqueClock += getID();
22    println("received <" + message.getSender() + ", "
23      + rlc + "> at " + uniqueClock);
24  }
25 }
```

Aufbau, Verbindung und Initialisierungen können vom vorigen Beispiel übernommen werden.

Vektoruhren Vektoruhren sind eine Alternative zu den Lamport-Uhren. Dabei verwaltet jeder Prozess einen Vektor mit den Zeitstempeln der anderen Prozesse nach folgenden Regeln:

VC1 Zunächst ist $V_i[j] := 0$ für $i, j = 1, 2, ..., N$.

VC2 Unmittelbar bevor p_i einem Ereignis einen Zeitstempel zuweist, setzt er $V_i[i] := V_i[i] + 1$.

VC3 p_i gibt jeder versendeten Nachricht den Wert $t := V_i$ mit.

VC4 Wenn p_i einen Zeitstempel t in einer Nachricht empfängt, setzt er

$$V_i[j] := max(V_i[j], t[j]) \text{ für } j = 1, 2, ..., N.$$

Diese komponenetenweise Bildung des Maximalwerts für zwei Vektor-Zeitstempel wird auch als *merge*-Operation bezeichnet. Bevor dem Empfang der Nachricht ein Zeitstempel zugewiesen wird, wird **VC2** angewendet.

Eine Implementierung dieses Verfahrens in DisASTer könnte wie folgt aussehen, wobei der Vektor noch mit `null` initialisiert wird, da beim Aufruf des Konstruktors noch nicht alle anderen Prozesse bekannt sind:

```
1 public class VectorClockAgent extends Agent {
2   private int[] v;
3
4   public VectorClockAgent() {
5     description = "VectorClockAgent " + getID();
6     v = null;
7   }
```

Auch in diesem Beispiel beschränkt sich die Ereignisspezifikation auf das Senden und den Empfang von Nachrichten. Zum Senden einer Nachricht wird die `init()`-Methode verwendet. Dort wird zunächst der Vektor initialisiert (**VC1**), sofern dies noch nicht geschehen ist. Die Initialisierung erfolgt in der Methode `initVector()`, die später vorgestellt wird. Nach der Initialisierung und vor dem Senden der Nachricht an den nächsten Nachbarn wird die eigene Uhr im Vektor laut **VC2** erhöht. Die Nachricht enthält laut **VC3** die eigene Vektoruhr. Um ein Ereignis zu simulieren, wird eine Textausgabe verwendet. Diese greift auf die Hilfsmethode `print(int[])` zu, die zur Darstellung der Vektoren verwendet und später beschrieben wird.

```
8   public void init(InitMessage message) {
9     initVector();
10
11    v[getID()]++;
12    send(v, getOthers()[0]);
13    println("sending Message to " + getOthers()[0]
14      + " at " + print(v));
15  }
```

Falls der Vektor beim Empfang einer Nachricht noch nicht initialisiert wurde, wird dies zunächst nachgeholt. Dann wird, nach Regel **VC4**, der eigene Vektor

v mit dem empfangenen Vektor vm verglichen und die einzelnen Zeiten auf das Maximum der beiden Vektoren gesetzt. Das eigentliche Ereignis stellt erneut eine Textausgabe dar.

```
16   public void onReception(Message message) {
17     initVector();
18
19     int[] vm = (int[]) message.getPayload();
20     int senderID = message.getSender().getID();
21
22     for (int i = 0; i < v.length; i++) {
23       v[i] = Math.max(v[i], vm[i]);
24     }
25     v[getID()]++;
26     println("received <" + message.getSender() + ", "
27       + print(vm) + "> at " + print(v));
28   }
```

Bei der Initialisierung des Vektors wird zunächst überprüft, ob dieser schon initialisiert wurde, sprich sein Wert noch `null` ist. Ist dies nicht der Fall wird er, nach Regel **VC1** mit 0 an allen Einträgen initialisiert.

```
29   private void initVector() {
30     if(v == null) {
31       v = new int[getOthers().length+1];
32       for (int i = 0; i < v.length; i++) {
33         v[i] = 0;
34       }
35     }
36   }
```

Um alle Werte des Vektors in einer angemessenen Form, $(v_1, v_2, ..., v_N)$, ausgeben zu können wird folgende Hilfsmethode verwendet:

```
37   private String print(int[] v) {
38     StringBuffer sb = new StringBuffer("(");
39     for (int i = 0; i < v.length; i++) {
40       if(i < v.length-1)
41         sb.append(v[i] + ",");
42       else
43         sb.append(v[i] + ")");
44     }
45     return new String(sb);
46   }
47 }
```

Auch dieses Beispiel lässt sich mit der Konfiguration der vorangegangenen ausführen, zwei oder mehr Agenten, die jeweils miteinander verbunden sind.

8.2 Koordination und Übereinstimmung

8.2.1 Verteilter wechselseitiger Ausschluss

Der verteilte wechselseitige Ausschluss wird in Abschnitt 3.3.1 beschrieben. Dabei geht es um das „kritische Abschnitt-Problem“, bei dem sich die Prozesse eines verteilten Systems darauf einigen müssen, welcher von ihnen den kritischen Abschnitt betreten darf.

Wechselseitiger Auschluss mit zentralem Server Beim wechselseitigen Ausschluss mit zentralem Server wird ein Prozess als zentraler Server gewählt, der von den anderen Prozessen Anfragen erhält, den kritischen Abschnitt zu betreten. Fallen mehr als eine Anfrage an, werden diese gepuffert und erhalten nacheinander die Zusage den kritischen Abschnitt zu betreten, sobald dieser wieder frei wurde.

Das Problem bei der Implementierung ist die Wahl des zentralen Servers. Damit diese Wahl nicht zusätzlich in den Algorithmus eingebaut werden muss, wird der erste Agent, der mit der ID 0, als zentraler Server festgelegt. Um ihn von den anderen Agenten herauszuheben bekommt er die Rolle `ROLE_MASTER` zugeschrieben. Dieser erhält den Puffer der Anfragen und eine boolsche Variable, die anzeigt, ob der kritische Abschnitt frei ist oder nicht. Auf die Implementierung des Servers in einer separaten Klasse wurde in diesem Beispiel noch verzichtet, wodurch es ein wenig unübersichtlich ist.

```
1  public class CentralServerExclusionAgent extends Agent {
2    private ArrayList buffer;
3    private boolean voted;
4
5    public CentralServerExclusionAgent() {
6      if(getID() == 0) {
7        description = "Server";
8        setRole(ROLE_MASTER);
9        buffer = new ArrayList();
10       voted = false;
11     }
12     else {
13       description = "CentralServerAgent " + getID();
14     }
15   }
```

Den Wunsch, den kritischen Abschnitt zu betreten, wird durch die Initialisierung über die `init()`-Methode ausgelöst. Dabei muss der Agent feststellen, wer der zentrale Server ist, um ihm seine Anfrage zu senden, bzw. falls der Agent selbst der zentrale Server ist, muss er feststellen, ob er selbst den kritischen Abschnitt betreten darf oder nicht:

```
16   public void init(InitMessage message) {
17     if(getState() == STATE_WORKING || getState() == STATE_WAITING)
```

```
18        return;
19      setState(STATE_WAITING);
20
21      if (getRole() != ROLE_MASTER) {
22        for (Iterator iter = connections.keySet().iterator();
23         iter.hasNext();) {
24          Agent agent = (Agent) iter.next();
25          if (agent.getID() == 0) {
26            send(new Request(agent));
27            return;
28          }
29        }
30      }
31      else {
32        if(buffer.isEmpty() && voted == false) {
33          setState(STATE_WORKING);
34          setTimeout(3);
35        }
36        else {
37          buffer.add(this);
38        }
39      }
40    }
```

Erhält der zentrale Server nun eine Anfrage, den kritischen Abschnitt zu betreten, überprüft er zunächst ob dieser frei ist und ob nicht schon Anfragen gestellt wurden. Ist dies nicht der Fall, kann er dem anfragenden Agenten sein OK geben und merkt sich, dass der kritische Abschnitt nun belegt ist. Ansonsten muss er die Anfrage zunächst puffern.

```
41    public void onReception(Request message) {
42      if(getRole() == ROLE_MASTER) {
43        if(buffer.isEmpty() && voted == false) {
44          voted = true;
45          send(new Response(message.getSender()));
46        }
47        else
48          buffer.add(message.getSender());
49      }
50    }
```

Der Nachrichtentyp `Response` wird für zwei Ereignisse verwendet. Zum einen sendet der Server einen `Response`, um dem Empfänger mitzuteilen, dass dieser den kritischen Abschnitt betreten darf. Zum anderen senden die Agenten, die den kritischen Abschitt verlassen, einen `Response` an den Server, um ihm mitzuteilen, dass der kritische Abschitt wieder betreten werden darf.

```
51    public void onReception(Response message) {
52      if(getRole() == ROLE_MASTER) {
53        if(buffer.isEmpty()) {
54          voted = false;
55        }
56        else {
57          Agent agent = (Agent) buffer.remove(0);
58          if(agent == this) {
59            setState(STATE_WORKING);
60            setTimeout(3);
```

```
61          }
62          else {
63            send(new Response(agent));
64            voted = true;
65          }
66        }
67      }
68      else {
69        setState(STATE_WORKING);
70        setTimeout(3);
71      }
72    }
```

Ein Timeout wird dazu verwendet, die Zeit zu simulieren, die ein Agent im kritischen Abschnitt verbringt. Nach Ablauf des Timeouts kann dieser dann dem Server mitteilen, dass er wieder frei ist. Falls sich der Server selbst im kritischen Abschnitt aufgehalten hat, muss er noch ausstehende Anfragen beantworten.

```
73    public void timeout(TimeoutMessage timeout) {
74      setState(STATE_NORMAL);
75      if(getRole() == ROLE_MASTER) {
76        if(buffer.isEmpty()) {
77          voted = false;
78        }
79        else {
80          Agent agent = (Agent) buffer.remove(0);
81          if(agent == this) {
82            setState(STATE_WORKING);
83            setTimeout(3);
84          }
85          else {
86            send(new Response(agent));
87            voted = true;
88          }
89        }
90      }
91      else {
92        for (Iterator iter = connections.keySet().iterator();
93         iter.hasNext();;) {
94          Agent agent = (Agent) iter.next();
95          if (agent.getID() == 0) {
96            send(new Response(agent));
97            return;
98          }
99        }
100     }
101   }
102 }
```

Voraussetzung diese Algorithmus ist, dass alle Agenten miteinander verbunden sind (CompleteTopology). Für den Server muss auch keine separate Klasse geladen werden, d.h. alle Agenten sind Instanzen dieser einen Klasse. Es spielt keine Rolle, welcher Agent wann initialisiert wird.

Ring basierender Ausschluss Um einen Zutritt zum kritischen Abschnitt in einer Ringstruktur zu erhalten, wird im in Abschnitt 3.3.1 beschriebenen Algorithmus ein Token um den Ring gesendet, der wie als Schlüssel fungiert und den Eintritt in den kritischen Abschnitt ermöglicht.

Bei dieser Implementierung wird der Schlüssel, also das Token, durch die Rolle eines Agenten definiert. Der Agent, der den Token besitzt, bekommt die Rolle `ROLE_MASTER`. Zu Begin des Algorithmus erhält der erste Agent, mit der ID 0, das Token. Damit die Agenten wissen, ob eine Anfrage besteht und sie bei Erhalt des Token diesen wieder weitersenden können, bekommt jeder Agent eine boolsche Variable, die diesen Zustand beschreibt.

```
1 public class RingExclusionAgent extends Agent {
2   private boolean gotRequest;
3
4   public RingExclusionAgent() {
5     description = "RingExcl-Agent " + getID();
6     if(getID() == 0)
7       setRole(ROLE_MASTER);
8   }
```

Der Wunsch, den kritischen Abschnitt zu betreten, wird durch die Initialisierung ausgelöst. Besitzt der initialisierte Agent bereits das Token, kann er den kritischen Abschnitt betreten. Andernfalls sendet er eine Anfrage über den Ring.

```
9   public void init(InitMessage message) {
10    if(getRole() != ROLE_MASTER) {
11      setState(STATE_WAITING);
12      sendToAll(new Request());
13    }
14    else {
15      setState(STATE_WORKING);
16      setTimeout(3);
17    }
18  }
```

Trifft eine Anfrage bei einem Agenten ein, der den Token selbst nicht besitzt, leitet er die Anfrage an den Ring weiter. Hält er sich selbst im kritischen Abschnitt auf, merkt er sich, dass eine Anfrage gestellt wurde. Besitzt der Agent das Token ohne selbst einen Abschnittswunsch zu haben, sendet er es, in Form einer Antwort (Response) weiter.

```
19  public void onReception(Request message) {
20    if(getState() != STATE_WORKING && getRole() == ROLE_MASTER) {
21      sendToAll(new Response());
22      setRole(ROLE_SLAVE);
23      gotRequest = false;
24    }
25    else if(getState() == STATE_WORKING) {
26      gotRequest = true;
27    }
28    else {
29      sendToAll(message);
30      gotRequest = false;
```

```
31      }
32    }
```

Erhält ein Agent einen `Response`, erhält er gleichzeitig das Token. Hegt dieser Agent selbst einen Eintrittswunsch, kann er den kritischen Abschnitt betreten. Andernfalls leitet er das Token wieder weiter.

```
33    public void onReception(Response message) {
34      setRole(ROLE_MASTER);
35      if(getState() == STATE_WAITING) {
36        setState(STATE_WORKING);
37        setTimeout(3);
38      }
39      else {
40        sendToAll(message);
41        setRole(ROLE_SLAVE);
42        gotRequest = false;
43      }
44    }
```

Der Ablauf eines Timeouts deutet dem Agenten an, dass er seine Arbeit im kritischen Abschnitt abgeschlossen hat und diesen wieder verlassen kann. Ist immer noch eine Anfrage eines anderen Agenten offen, sendet er das Token auf den Ring.

```
45    public void timeout(TimeoutMessage timeout) {
46      setState(STATE_NORMAL);
47      if(gotRequest) {
48        sendToAll(new Response());
49        setRole(ROLE_SLAVE);
50        gotRequest = false;
51      }
52    }
53  }
```

Dieser Algorithmus benötigt auch nur Agenten dieser Klasse. Es sollten mindestens zwei, besser noch mehr als drei Agenten eingebunden werden, die den Topologiemanager `TokenRing` verwenden. Welcher Agent dabei inittialisiert wird, ist nebensächlich.

Gegenseitiger Ausschluss von Ricart und Agrawala Die Lösung des wechselseitigen verteilten Ausschlusses von Ricart und Agrawala wurde für ein Netzwerk mit Multicast-Funktionalität entwickelt. Er lässt sich wie folgt beschreiben:

- Eintrittswunsch über Multicast an alle anderen Prozesse senden und auf deren Antworten warten.
- Nachdem eine Antwort von jedem anderen Prozess eingetroffen ist, kann der kritische Abschnitt betreten werden.
- Beim Eintreffen einer Anfrage zum Eintritt in den kritischen Abschnitt:

1. falls selbst kein Eintrittswunsch vorliegt, Einverständnis zum Eintritt senden.
2. falls selbst Eintrittswunsch vorliegt und Zeitstempel der Anfrage kleiner als der der eigenen Anfrage, Einverständnis senden. Falls der eigene Eintrittswunsch vor der empfangenen Anfrage erfolgte, wird die Anfrage gepuffert.
3. falls selbst im kritischen Abschnitt, Anfragen puffern und nach Austritt Einverständnisse aller gepufferter Anfragen senden.

Ein Agent, der den Algorithmus von Ricart und Agrawala implementiert, muss daher einige Informationen speichern. Zunächst einmal muss er über einen Puffer verfügen, in dem Anfragen anderer Agenten gespeichert werden. Er braucht einen Zähler, um die Antworten der anderen Agenten, auf seine Anfragen, aufzunehmen, eine logische Uhr, um das Verfahren fair ausführen zu können und einen Wert, in dem er die Zeit seiner eigenen Anfrage speichern kann.

```
1  public class MutualExclusionAgent extends Agent {
2    private ArrayList buffer;
3    private int answers;
4    private int clock;
5    private int sentTime;
6
7    public MutualExclusionAgent(){
8      setDescription("MutEx-Agent " + getID());
9      clock = 0 + getID();
10     sentTime = 0;
11     buffer = new ArrayList();
12     setState(STATE_NORMAL);
13   }
```

Der Eintrittswunsch wird durch die Initialisierung des Agenten ausgelöst. Dabei erhöht er seine logische Uhr und sendet eine Anfrage an die anderen Agenten. Er merkt sich den Zeitpunkt, an dem er die Anfrage gestellt hat und setzt den Zähler der Antworten auf 0.

```
16   public void init(InitMessage message){
17     clock += 10;
18     sendToAll(new Request(null, new Integer(clock)));
19     setState(STATE_WAITING);
20     sentTime = clock;
21     answers = 0;
22   }
```

Erhält ein Agent eine Anfrage, muss er seinen momentanen Zustand überprüfen. Hegt er selbst einen Eintrittswunsch, vergleicht er die Sendezeit der Anfrage mit seiner eigenen. Ist diese kleiner als seine eigene, muss er dem Sender Vorrang lassen und sendet ihm sein OK, ansonsten puffert er die Anfrage. Will der Agent selbst nicht den kritischen Abschnitt betreten, sendet er unmittelbar sein

OK. Befindet sich der Agent im kritischen Abschnitt, wird die Anfrage ebenfalls gepuffert.

```
23   public void onReception(Request request){
24     int rlc = ((Integer) message.getPayload()).intValue();
25     clock = Math.max(clock, rlc);
26     clock += 10;
27     clock -= (clock % 10);
28     clock += getID();
29
30     switch(getState()){
31       case STATE_WAITING :
32         if( ((Integer) request.getPayload() ).intValue()
33           < sentTime ){
34           send(new Response(request.getSender(),
35             new Integer(clock)));
36         }
37         else{
38           buffer.add(request.getSender());
39         }
40         break;
41       case STATE_NORMAL :
42         send(new Response(request.getSender(),
43           new Integer(clock)) );
44         break;
45       case STATE_WORKING :
46         buffer.add(request.getSender());
47         break;
48     }
49   }
```

Der Erhalt einer Antwort erhöht zunächst die logische Uhr des Empfängers. dann wird die Anzahl der erhaltenen Antworten inkrementiert. Hat der Agent mit dieser Antwort von allen anderen Agenten eine Zusage bekommen, kann er den kritischen Abschnitt betreten.

```
45   public void onReception(Response response){
46     int rlc = ((Integer) message.getPayload()).intValue();
47     clock = Math.max(clock, rlc);
48     clock += 10;
49     clock -= (clock % 10);
50     clock += getID();
51
52     answers++;
53     if(answers == getOthers().length){
54       setState(STATE_WORKING);
55       setTimeout(5);
56     }
57   }
```

Nach Ablauf eines Timeouts wird der kritische Abschnitt wieder verlassen. Befinden sich noch Anfragen im Puffer, werden diese alle beantwortet.

```
53   public void timeout(TimeoutMessage message){
54     clock+=10;
55     setState(STATE_NORMAL);
56     answers = 0;
57     for( Iterator i = buffer.iterator(); i.hasNext();){
58       Agent agent = (Agent) i.next();
```

```
59        send(new Response(agent, new Integer(clock)));
60        i.remove();
61      }
62    }
63 }
```

Dieser Algorithmus arbeitet ausschließlich mit Agenten dieser Klasse. Die eingebunden Agenten sollten über die `CompleteTopology` miteinander verbunden werden. Interessant wäre in diesem Fall die gleichzeitige Initialisierung mehrerer Agenten.

Wahl-Algorithmus von Maekawa Maekawas Algorithmus ähnelt dem von Ricart und Agrawala. Er teilt die Prozesse allerdings in Teilgruppen auf, um die Kommunikation zu reduzieren. Die genaue Beschreibung des Algorithmus befindet sich in Abschnitt 3.3.1 ab Seite Seite 19.

Ein Agent, der diesen Algorithmus implementiert, besitzt eine boolsche Variable, die anzeigt, ob er schon einem anderen Agenten die Zusage gemacht hat, den kritischen Abscnitt zu betreten. Des weiteren hat er einen Puffer, um weitere Anfragen zu speichern und einen Zähler, der die eingegangenen Antworten zählt.

```
1 public class MaekawaAgent extends Agent {
2   private boolean voted;
3   private ArrayList buffer;
4   private int answers;
5
6   public MaekawaAgent() {
7     description = "Maekawa " + getID();
8     buffer = new ArrayList();
9     voted = false;
10  }
```

Mit der Initialisierung wird der Eintrittswunsch ausgelöst. Dabei wird eine Anfrage an alle Gruppenmitglieder des Agenten gesendet.

```
11  public void init(InitMessage message) {
12    setState(STATE_WAITING);
13    answers = 0;
14    sendToAll(new Request());
15  }
```

Bei Erhalt einer solchen Anfrage überprüft der Empfänger, ob er schon einem anderen Agenten die Zusage gemacht hat oder ob er sich selbst im kritischen Abschnitt befindet. Ist dies der Fall, puffert er die Anfrage, ansonsten sendet er sein OK zurück und merkt sich, dass er seine Stimme bereits abgegeben hat.

```
16  public void onReception(Request message) {
17    if(voted == true || getState() == STATE_WORKING) {
18      buffer.add(message.getSender());
19    }
20    else {
21      send(new Response(message.getSender(), null));
22      voted = true;
```

```
23      }
24    }
```

Erhält ein Agent eine Antwort auf seine Anfrage, erhöht er deren Anzahl. Sind mit dieser Antwort alle eingetroffen, kann er den kritischen Abschnitt betreten.

```
25    public void onReception(Response message) {
26      answers++;
27      if(answers == getOthers().length) {
28        setState(STATE_WORKING);
29        setTimeout(3);
30      }
31    }
```

Der Erhalt einer einfachen Nachricht deutet an, dass der Sender dieser Nachricht den kritischen Abschnitt verlassen hat und dieser wieder frei ist. Hat der Empfänger noch Anfragen in dem Puffer, wird die erste beantwortet. Dabei muss sich der Agent erneut merken, dass er jetzt seine Stimme abgegeben hat.

```
32    public void onReception(Message message) {
33      if(buffer.isEmpty()) {
34        voted = false;
35      }
36      else {
37        send(new Response((Agent)buffer.remove(0), null));
38        voted = true;
39      }
40    }
```

Nach einem Timeout verlässt ein Agent wieder den kritischen Abschnitt und teilt dies seinen Gruppenmitglieder über eine einfache Nachricht mit.

```
41    public void timeout(TimeoutMessage timeout) {
42      setState(STATE_NORMAL);
43      sendToAll(new Message());
44    }
45  }
```

Auch für diesen Algorithmus wird nur eine Klasse verwendet. Die Verbindungen zwischen den einzelnen Agenten sollte in diesem Fall allerdings manuell durchgeführt werden, um die Gruppen herstellen zu können, denen die Agenten zugeordnet sind.

8.2.2 Wahl eines Anführers

Die Wahl eines Anführers ist ein Problem, dass bereits von einigen vorangegangenen Algorithmen verwendet wurde, um beispielsweise einen zentralen Server zu bestimmen. Eine genaue Beschreibung der Problematik ist in Abschnitt 3.3.2 zu finden.

Ringbasierte Wahl Chang und Roberts haben ein Verfahren entwickelt, um einen Anführer in einer Ringstruktur festzulegen.:

- Um die Wahl zu initiieren, wird eine Wahlnachricht mit der eigenen ID an den nächsten Prozess gesendet. Zusätzlich merkt sich der Prozess, dass er bereits Teilnehmer einer Wahl ist.
- Bei Erhalt einer Wahlnachricht überprüft der Empfänger seine ID mit der der Nachricht.
  1. Ist die eigene ID kleiner als die empfangene, sendet er die Nachricht an den nächsten Prozess weiter und setzt sich als Teilnehmer.
  2. Ist die eigene ID größer und der Empfänger noch kein Teilnehmer einer Wahl, trägt er seine eigene ID in die Nachricht ein, sendet sie weiter und setzt sich als Teilnehmer. Ansonsten wird die Nachricht verworfen.
  3. Ist die empfange ID gleich der eigenen, dann ist der Empfänger der neue Anführer. Dieser setzt sich als Nicht-Teilnehmer und sendet eine Wahlergebnisnachricht mit seiner eigenen ID an den nächsten Prozess.
- Bei Empfang einer Wahlergebnisnachricht merkt sich der Empfänger die ID des neuen Anführers, markiert sich selbst als Nicht-Teilnehmer und sendet die Nachricht an den nächsten Prozess weiter, falls er nicht selbst Anführer ist.

Bei der Implementierung dieses Algorithmus werden zunächst einmal die Zustandskonstanten angepasst. Es werden die Konstanten `MEMBER` und `NO_MEMBER` eingeführt, die anzeigen sollen, ob ein Agent bereits bei einer Wahl beteiligt ist oder nicht. Die ID des zukünftigen Anführers wird in der Variablen `leader` gespeichert und mit -1 initialisiert.

```
1  public class RingLeaderAgent extends Agent {
2    public static final int MEMBER = STATE_WORKING;
3    public static final int NO_MEMBER = STATE_NORMAL;
4
5    private int leader;
6
7    public RingLeaderAgent() {
8      description = "Ringleader " + getID();
9      setState(NO_MEMBER);
10     leader = -1;
11   }
```

Durch die Initialisierung eines Agenten wird zugleich eine neue Wahl initiiert. Dabei sendet der Agent eine Anfrage mit seiner Kennung über den Ring.

```
12   public void init(InitMessage message) {
13     sendToAll(new Request(null, new Integer(getID())));
14     setState(MEMBER);
15     setRole(ROLE_SLAVE);
16   }
```

Erhält ein Agent eine Anfrage, vergleicht er die empfangene ID mit seiner eigenen. Ist seine eigene größer und er noch nicht Teilnehmer einer Wahl, sendet er diese anstatt der empfangenen an den Ring weiter. Ist seine eigene kleiner als die empfangene, leitet er die Nachricht weiter. In beiden Fällen setzt er sich als Teilnehmer einer Wahl. Beinhaltet die empfangene Nachricht seine eigene ID, bedeutet das, dass er der neue Anführer ist und er teilt seine Erkenntnis über einen `Response` dem Ring mit.

```
17   public void onReception(Request message) {
18     int id = ((Integer)message.getPayload()).intValue();
19
20     if(getID() < id) {
21       sendToAll(new Request(null, new Integer(id)));
22       setState(MEMBER);
23     }
24     else if(getID() > id && getState() != MEMBER) {
25       sendToAll(new Request(null, new Integer(getID())));
26       setState(MEMBER);
27     }
28     else if(getID() == id){
29       sendToAll(new Response(null, new Integer(getID())));
30       setState(NO_MEMBER);
31       setRole(ROLE_MASTER);
32       leader = getID();
33     }
34   }
```

Bei Erhalt einer Antwort trägt der Empfänger die empfangene Kennung als die des jetzigen Anführers ein. Falls er nicht selbst der Anführer ist, leitet er die Nachricht an den Ring weiter.

```
35   public void onReception(Response message) {
36     leader = ((Integer)message.getPayload()).intValue();
37
38     if(getID() != leader) {
39       sendToAll(new Response(null, new Integer(leader)));
40       setState(NO_MEMBER);
41     }
42   }
43 }
```

Um diesen Algorithmus korrekt ausführen zu können, wird die `TokenRing`-Topologie benötigt. Die Anzahl der Agenten sollte mindestens zwei, besser noch mehr als drei betragen. Interessant ist in diesem Fall die Initialisierung des Agenten, der nach dem Agenten mit der höchsten Kennung kommt. Um den Algorithmus noch interessanter zu gestalten, kann die Reihenfolge der Agenten durcheinandergebracht werden, bevor sie mit der `TokenRing`-Topologie verbunden werden,

da dadurch die IDs nicht in aufsteigender Reihenfolge, sondern willkürlich verteilt werden.

Alternatives Ringverfahren Das alternative Ringverfahren soll den worst-case der einfachen Ring basierten Wahl abfangen und die Nachrichtenanzahl reduzieren. Hierzu wird anstatt nur eine Kennung über den Ring zu senden, eine Liste aller IDs gesammelt und der Initiator der Wahl bestimmt davon das Maximum.

Der Aufbau und der Konstruktor ist äquvalent zum vorangegangenen Beispiel.

```
1 public class RingLeaderAgent extends Agent {
2   public static final int MEMBER = STATE_WORKING;
3   public static final int NO_MEMBER = STATE_NORMAL;
4
5   private int leader;
6
7   public RingLeaderAgent() {
8     setDescription("RingLeader " + getID());
9     setState(NO_MEMBER);
10    leader = -1;
11  }
```

Bei der Initiierung einer Wahl wird in diesem Fall aber keine einzelne ID, sondern eine Liste losgeschickt. In dieser wird zunächst nur die Kennung des Initiators eingetragen.

```
12  public void init(InitMessage message) {
13    ArrayList ids = new ArrayList();
14    ids.add(new Integer(getID()));
15    send(new Request(Agent.BROADCAST, ids));
16    setState(MEMBER);
17  }
```

Erhält ein Agent diese Liste, trägt er seine eigene Kennung ein, merkt sich, dass er an einer Wahl teilnimmt und sendet die Liste an den Ring weiter. Hat der Agent diese Wahl selbst initiiert, wertet er die Liste aus, indem er das Maximum bestimmt. Das Ergebnis sendet er in Form eines Response über den Ring.

```
18  public void onReception(Request message) {
19    ArrayList ids = (ArrayList) message.getPayload();
20    if( ((Integer)ids.get(0)).intValue() == getID() ) {
21      setState(NO_MEMBER);
22      int max = getID();
23      for (Iterator iter = ids.iterator(); iter.hasNext();) {
24        Integer id = (Integer) iter.next();
25        max = Math.max(max, id.intValue());
26      }
27      leader = max;
28      send(new Response(BROADCAST, new Integer(max)));
29      if(max == getID()) {
30        setRole(Agent.ROLE_MASTER);
31      }
32    }
33    else {
```

```
34        if(getState() == MEMBER)
35          return;
36        setState(MEMBER);
37        ids.add(new Integer(getID()));
38        send(new Request(BROADCAST, ids));
39      }
40    }
```

Erhält ein Agent ein `Response`, merkt er sich den neuen Anführer und sendet die Nachricht an den Ring weiter. Ist er selbst der neue Anführer, setzt er seine Rolle auf `ROLE_MASTER`

```
41    public void onReception(Response message) {
42      int max = ((Integer)message.getPayload()).intValue();
43      if(leader != max) {
44        send(new Response(BROADCAST, message.getPayload()));
45        setState(NO_MEMBER);
46      }
47      if(max == getID()) {
48        setRole(Agent.ROLE_MASTER);
49        leader = max;
50        setState(NO_MEMBER);
51      }
52    }
53  }
```

Dieser Algorithmus sollte wie der im vorangegangenen Beispiel konfiguriert werden.

Bully-Algorithmus Der Bully-Algorithmus von Garcia-Molina setzt voraus, dass alle Prozesse die Kennungen der anderen Prozesse kennen. Der Algorithmus arbeitet wie folgt:

1. Merkt ein Prozess, dass der derzeitige Anführer ausgefallen ist, startet er eine Wahl, sofern er nicht die nunmehr höchste Kennung besitzt.

2. Um eine Wahl zu starten, werden Wahlnachrichten an alle Prozesse mit einer höheren ID gesendet und auf dessen Antwort gewartet.

3. Trifft nach einem Timeout keine Antwort von einem Prozess ein, gilt dieser als ausgefallen.

4. Beim Eintreffen einer Wahlnachricht initiiert der Empfänger eine eigene Wahl (2.).

5. Stellt ein Prozess fest, dass er die höchste ID im System hat, z.B. dadurch, dass er keine Antwort von einem Prozess mit höherer ID erhalten hat, sendet er eine Koordinatornachricht an alle anderen Prozesse um ihnen dies mitzuteilen.

Bei der Implementierung eines Agenten, der einen Anführer mit Hilfe des Bully-Algorithmus bestimmen soll, wird zunächst eine Konstante `TIMEOUT` festgelegt, die festlegt, wie lange ein Agent bzw. dessen Antwort Zeit hat, bevor dieser als ausgefallen gerechnet wird. Die ausgefallenen Agenten sollen in der Liste `corrupted` aufgeführt werden.

```
1 public class BullyAgent extends Agent {
2   public static final int TIMEOUT = 5;
3
4   private ArrayList corrupted;
5   private int inits;
6   private Agent leader;
7
8   public BullyAgent() {
9     description = "Bully " + getID();
10    corrupted = new ArrayList();
11  }
```

Die Initialisierung eines Agenten ist gleichzusetzen mit der Initiierung der Wahl. Im Gegensatz zu den bisherigen Verfahren kann die Wahl eines Agenten auch durch andere Agenten erfolgen. Der Sender einer solchen Initialisierungsnachricht bekommt dann eine Antwort von ihrem Empfänger, um diesem mitzuteilen, dass er nicht ausgefallen ist. Falls der Agent noch keine eigene Wahl initiiert hat und er selbst nicht der neue Anführer ist, beginnt er eine neue Wahl und sendet eine Initialisierungsnachricht an alle Agenten mit einer höheren ID. Diese trägt er zunächst in die Liste der ausgefallenen Prozesse ein, bis sie ihm eine Antwort auf die gerade gesendete Nachricht schicken. Zusätzlich zur Initialisierung wird ein Timeout ausgelöst, der den gerade angeschriebenen Agenten enthält, damit der Agent überprüfen kann, ob er schon eine Antwort nach dem entsprechenden Timeout bekommen hat. Die Variable `inits` zeigt an, wie viele andere Agenten initialisiert wurden. Stellt er bei dieser Gelegenheit fest, dass kein Agent eine höhere Kennung hat als er selbst, ist er der neue Anführer und teilt dies allen anderen Agenten mit.

```
12  public void init(InitMessage message) {
13    if(message.getSender() != null) {
14      send(new Response(message.getSender()));
15    }
16
17    if(getState() == STATE_WAITING || leader == this)
18      return;
19
20    setState(STATE_WAITING);
21    inits = 0;
22    leader = null;
23
24    boolean isLeader = true;
25    for (int i = 0; i < getOthers().length; i++) {
26      Agent other = getOthers()[i];
27      if(other.getID() > getID()) {
28        isLeader = false;
29        send(new InitMessage(other));
```

```
30          setTimeout(other, TIMEOUT);
31          corrupted.add(other);
32          inits++;
33        }
34      }
35
36      if(isLeader) {
37        leader = this;
38        setRole(ROLE_MASTER);
39        sendToAll(new Message(null, leader));
40        setState(STATE_NORMAL);
41      }
42    }
```

Erhält ein Agent eine Antwort auf eine Initialisierung, kann er den Sender der Nachricht aus der Liste der ausgefallenen Prozesse streichen.

```
43    public void onReception(Response message) {
44      corrupted.remove(message.getSender());
45    }
```

Empfängt ein Agent eine einfache Nachricht, weiß er, dass der neue Anführer der Sender dieser Nachricht ist.

```
46    public void onReception(Message message) {
47      leader = message.getSender();
48      setState(STATE_NORMAL);
49      setRole(ROLE_SLAVE);
50    }
```

Für jede gesendete Initialisierung eines Agenten mit höherer Kennung läuft ein Timeout. Inhalt des Timeouts ist die Referenz auf den Agenten, der in einer bestimmten Zeit eine Antwort senden muss, um nicht als ausgefallen zu gelten. Sind alle Timeouts abgelaufen hat `inits` den Wert 0. Befinden sich zusätzlich noch Agenten in der Liste der Ausgefallenen, ist dieser Agent der neue Anführer und teilt dies den anderen Agenten mit.

```
51    public void timeout(TimeoutMessage timeout) {
52      if(getState() == STATE_NORMAL)
53        return;
54
55      inits--;
56      if(inits == 0 && corrupted.size() > 0) {
57        setRole(ROLE_MASTER);
58        leader = this;
59        sendToAll(new Message(null, leader));
60      }
61    }
62  }
```

Dieser Algorithmus benötigt die `CompleteTopology`. Um den Algorithmus interessanter zu gestalten und evtl. einige Ausfälle zu simulieren, könnte die Konstante TIMEOUT erniedrigt oder Nachrichten während des Ablaufs gelöscht werden.

8.2.3 Übereinstimmungsprobleme

Die Übereinstimmungsprobleme werden in Abschnitt 3.3.4 beschrieben. Diese Probleme beschäftigen sich mit der Übereinstimmung korrekt laufender Prozesse, trotz Einfluss von fehlerhaften.

Problem der byzantinischen Generäle Das Problem der byzantinischen Generäle ist wohl das bekannteste Beispiel für Übereinstimmungsprobleme. Es geht dabei um die Auswertung eines Befehls von einem Kommandeur durch mehrere Generäle. Dabei kann unter den Generälen ein Verräter sein oder der Kommandeur selbst kann fehlerhaft arbeiten.

In dieser Implementierung wird zum ersten Mal mehr als eine Klasse verwendet und zwar eine für die loyalen Generäle und eine für die Verräter. Bei beiden Klassen wird der Kommandeur durch den Agenten repräsentiert, der die kleinste Kennung, ID ist 0, besitzt. Zunächst wird die Klasse für die loyalen Generäle besprochen. In dieser findet sich eine Liste der gesammelten Befehle (`commands`).

```
1  public class ByzantinicGeneral extends Agent {
2    private ArrayList commands;
3
4    public ByzantinicGeneral() {
5      if(getID() == 0) {
6        description = "Commander";
7        setRole(ROLE_MASTER);
8      }
9      else {
10       description = "General " + getID();
11       commands = new ArrayList();
12     }
13   }
```

Die Initialisierung besitzt nur beim Kommandeur eine Wirkung. Dadurch versendet dieser seinen Befehl an alle seine Generäle.

```
16   public void init(InitMessage message) {
17     if(getRole() == ROLE_MASTER) {
18       Integer command = new Integer(0);
19       sendToAll(new Message(null, command));
20       println("sending command " + command);
21     }
22   }
```

Erhält ein General einen Befehl vom Kommandeur, nimmt er diesen zunächst in seine Liste mit den empfangenen Befehlen auf und sendet ihn an alle anderen Generäle.

```
23   public void onReception(Message message) {
24     commands.add(message.getPayload());
25     for (int i = 0; i < getOthers().length; i++) {
26       Agent general = getOthers()[i];
27       if(general.getRole() != ROLE_MASTER) {
28         send(new Response(general, message.getPayload()));
```

```
29        }
30      }
31    }
```

Nach Erhalt eines Befehls von einem anderen General wird der Befehl der Liste hinzugefügt. Stellt der Agent fest, dass er von allen Generälen den Befehl empfangen hat, führt er den Befehl aus, den er am häufigsten erhalten hat. Um festzustellen, welcher Befehl das ist, wird die Hilfsmethode avg() verwendet.

```
32    public void onReception(Response message) {
33      println("got " + message.getSender().getID() + ":"
34        + message.getPayload());
35      commands.add(message.getPayload());
36      if(commands.size() == getOthers().length) {
37        println("Doing command " + avg());
38      }
39    }
```

Diese Methode soll den Befehl berechnen, den ein Agent durchzuführen hat. Dabei wird gezählt, wie häufig ein Befehl empfangen wurde. Der Befehl, der am häufigsten empfangen wurde, wird ausgeführt. Wenn kein Befehl die absolute Mehrheit bekommen hat, wird der Standardbefehl -1 zurückgeliefert.

```
40    public int avg() {
41      HashMap valueCounter = new HashMap();
42      for (Iterator i = commands.iterator(); i.hasNext();) {
43        Integer command = (Integer) i.next();
44        if(!valueCounter.containsKey(command))
45          valueCounter.put(command, new Integer(0));
46        int counter =
47          ( (Integer) valueCounter.get(command)).intValue();
48        valueCounter.put(command, new Integer(counter+1));
49      }
50      Iterator i = valueCounter.keySet().iterator();
51      Integer max = (Integer)i.next();
52      boolean equal = false;
53      while( i.hasNext() ) {
54        Integer command = (Integer) i.next();
55        int commandValue =
56          ( (Integer) valueCounter.get(command)).intValue();
57        int maxValue =
58          ( (Integer) valueCounter.get(max)).intValue();
59        if(commandValue > maxValue) {
60          max = command;
61          equal = false;
62        }
63        else if(commandValue == maxValue)
64          equal = true;
65      }
66      if(equal)
67        return -1;
68      else
69        return max.intValue();
70    }
71  }
```

Die Implementierung eines Verräters sieht dagegen ein wenig einfacher aus.

Hierbei fällt die Liste der Befehle weg, da sich ein Verräter sowieso nicht um das schert, was die anderen sagen.

```
1 public class ByzantinicTraitor extends Agent {
2   public ByzantinicTraitor() {
3     if(getID() == 0) {
4       description = "Commander (T)";
5       setRole(ROLE_MASTER);
6     }
7     else {
8       description = "General (T) " + getID();
9     }
10  }
```

Bei der Initialisierung eines verräterischen Kommandeurs, sendet dieser jedem General einen anderen Befehl.

```
11    public void init(InitMessage message) {
12      if(getRole() == ROLE_MASTER) {
13        for (int i = 0; i < getOthers().length; i++) {
14          Agent general = (Agent) getOthers()[i];
15          send(new Message(general, new Integer(i)));
16          println("sending command " + i + " to " + general);
17        }
18      }
19    }
```

Erhält ein verräterischer General einen Befehl des Kommandeurs, sendet er jedem anderen General einen anderen Befehl, nur nicht den, den er von Kommandeur erhalten hat.

```
20    public void onReception(Message message) {
21      int command = ( (Integer) message.getPayload()).intValue();
22      for (int i = 0; i < getOthers().length; i++) {
23        Agent general = getOthers()[i];
24        if(general.getRole() != ROLE_MASTER) {
25          send(new Response(general, new Integer(command + i)));
26          println("sending wrong command " + (command + i)
27            + " to " + general);
28        }
29      }
30    }
```

Diese Methode wird benötigt, da ankommende Antworten sonst in der normalen `onReception(Message)`-Methode landen würden. Die ankommenden Befehle der anderen Generäle werden von einem Verräter einfach ignoriert.

```
31    public void onReception(Response message) {
32      println("ignoring command " + message + " from "
33        + message.getSender());
34    }
```

Anders als bei allen anderen Beispielen werden für dieses zwei Klassen benötigt. Diese sollten nach den Vorgaben des Algorithmus eingesetzt werden, d.h. $3f + 1$ Agenten, wobei f die Anzahl der Verräter ist. Aus Testzwecken können

natürlich auch Szenarien mit mehr Verrätern erstellt werden. Die Agenten müssen über die `CompleteTopology` miteinander verbunden werden.

9 Verwandte Projekte

Dieses Kapitel beschäftigt sich mit der Untersuchung bereits bestehender Projekte, welche die Problematik, verteilte Algorithmen zu simulieren, bereits behandelt haben. Eine solche Recherche hilft, das eigentliche Problem genauer zu analysieren und sich mit Techniken vertraut zu machen, die bereits für das Lösen dieser Aufgabe eingesetzt werden. Nach einer genauen Untersuchung der Projekte lässt sich feststellen, wieweit diese Techniken als sinnvoll betrachtet werden können und welche in diesem Entwurf einen Platz finden werden.

9.1 Vorstellung der Projekte

Für das *DAP*-Projekt wurde bereits eine solche Analyse durchgeführt, bei der einige Bibliotheken und Anwendungen und deren Eigenschaften beschrieben werden. Hierbei handelt es sich zum Teil um Programme, die ebenfalls verteilte Algorithmen simulieren, andere beschränken sich auf die Darstellung von Netzwerkprotokollen. Neben *DAP* selbst werden bei Zaroliagis *ns*, *IOA*, *DSP*, *YACSIM* und *SimUTC* vorgestellt (vgl. [Zaroliagis02], Seite 3ff). Mit *Lydian* und *DAJ* wurden noch zwei weitere davon unabhängige Projekte untersucht. In den folgenden Abschnitten werden diese Programme kurz vorgestellt, um später ihre Vor- und Nachteile zu diskutieren.

ns - Network Simulator *ns*, was für Network Simulator steht, beschränkt sich auf die Simulation von Netzwerkprotokollen wie TCP, um Probleme wie Routing oder Multicast zu simulieren und zu testen. Zudem können kabellose Netzwerke (wireless network) simuliert werden, sei es lokal oder über Satellit. Um ein solches Netzwerk für *ns* zu erzeugen muss eine eigene Skriptsprache verwendet werden. Bei dieser Scriptsprache handelt es sich um OTcl, eine objektorientierte Erweiterung der im MIT entwickelten Tcl. Animation und Simulation verlaufen bei *ns* nicht zeitgleich. Zunächst wird die Simulation durchgeführt und dabei eine Log-Datei erstellt. Die in dieser Datei eingetragenen Ereignisse werden dann in der Animation verwendet und dargestellt (vgl. auch [McCanne, Floyd 03]).

IOA - I/O Automata *IOA* stellt eine formale Sprache dar, mit der verteilte Systeme oder Algorithmen modelliert werden können. Mit dem *IOA*-Toolkit werden Programme zur Simulation, zur Verifikation und zum Testen ausgeliefert. Zusätzlich kann Java-Code erzeugt werden, um die beschriebenen Systeme und

Algorithmen in einem realen Netzwerk einzusetzen (vgl. auch [Kaynar03]).

DSP - Distributed Systems Platform Die *Distributed Systems Platform* (*DSP*) wurde entwickelt, um verteilte Algorithmen zu implementieren, zu simulieren und zu testen. Die Topologie und das Verhalten des Algorithmus werden in einer eigenen Skriptsprache, der DSPL (DSP-Language), beschrieben. Die Simulation geschieht über eine Client/Server-Architektur, bei der der Client den in der DSPL verfassten Algorithmus an einen Server schickt. Dieser kompiliert das übergebene Programm, hält diesen Vorgang in seiner Datenbank fest und führt die Simulation aus, die dann beim Client dargestellt wird (vgl. auch [Terrevoli97]).

YACSIM *YACSIM*, Yet Another CSIM, ist eine Erweiterung der Sprache C. Es handelt sich dabei um eine Bibliothek, die Subroutinen und Datenstrukturen enthält, um ereignisgesteuerte oder prozessorientierte Simulationen zu erzeugen. Unter Zuhilfenahme dieser Komponenten können so verschiedene Simulationen nach dem Baukastenprinzip entwickelt werden. Die daraus resultierende Anwendung enthält gleichzeitig die Simulation und deren Laufzeitumgebung (vgl. auch [Jump93], Seite 4).

SimUTC *SimUTC* wurde entwickelt um Zeitsynchronisationsprobleme zu lösen. Algorithmen werden mit C++Sim, einer C++-Erweiterung, die SIMULA-Eigenschaften mitbringen soll, programmiert. Für seine prozessorientierte Simulation verwendet SimUTC Threads (vgl. auch [Weiss et. al. 99]).

DAP - Distributed Algorithms Platform *DAP* ist eine Erweiterung der LEDA-Bibliothek und wurde entworfen, um verteilte Algorithmen zu implementieren, zu simulieren und zu testen. Für die Implementierung wird C++ verwendet. Die Einrichtung und Simulation erfolgt über eine grafische Benutzeroberfläche, zu der unter anderem ein Topologieeditor gehört. Mit Hilfe einer zusätzlich beiliegende Bibliothek können die Algorithmen in einem realen Netzwerk eingesetzt werden.

Lydian *Lydian* wird für Lernzwecke eingesetzt. Hauptaufgabe dieses Programms ist die Simulation und Visualisierung von verteilten Algorithmen. Diese werden über eine eigene Beschreibungssprache definiert. *Lydian* setzt auf einige externe Bibliotheken wie z.B. LEDA, POLKA oder DIAS auf, die die Aufgaben der Netzwerkstrukturen oder die Darstellung der Algorithmen übernehmen (vgl. [Lydian]).

DAJ - Distributed Algorithms in Java *DAJ* wurde an der Universität Linz entwickelt. Es handelt sich um eine Java-Bibliothek, um verteilte Algorithmen zu implementieren, zu simulieren, zu visualisieren und zu testen. In der Bibliothek finden sich Klassen zur Erzeugung eines verteilten Algorithmus. Zudem wird eine Umgebung mitgeliefert, in der diese Algorithmen ablaufen können. Die Ausführung kann lokal als Applikation oder über das Web als Applet erfolgen. In *DAJ* erstellte Systeme werden durch einen Graphen, der aus Knoten und Kanten besteht, beschrieben. Die Knoten haben die Möglichkeit, Nachrichten an andere Knoten zu senden, um einen Algorithmus zu durchlaufen. Das macht *DAJ* zu einem nachrichtengesteuerten System (vgl. [Schreiner98]).

ViSiDiA *ViSiDiA* ist ein Werkzeug zur Implementierung, Simulation, Visualisierung und zum Testen von verteilten Algorithmen. Das Programm ist in und für Java geschrieben. Es untergliedert sich in zwei Teile:

1. einem Grapheditor, in dem die einzelnen Prozesse angeordnet und deren Verbindung gelegt werden.

2. einer Simulationsumgebung, in der unterschiedliche Algorithmen für die angelegte Topologie ausgewählt und simuliert werden können. Dabei helfen Funktionen wie Start, Stopp, Pause oder Reset. Die Simulationsgeschwindigkeit kann ebenfalls bestimmt werden und einige Informationsfenster geben einen zusätzlichen Überblick über den Zustand des Systems und der Umgebung, so z.B. ein Thread-Zähler.

Das Programm verwendet Threads für die Realisierung der einzelnen Prozesse (vgl. [Bauderon et. al. 03]).

9.2 Zusammenfassung und Vergleich

Nach der Untersuchung der einzelnen Projekte werden im folgenden deren Eigenschaften erörtert, welche Vor- und Nachteile sie haben und welche Gemeinsamkeiten zwischen den einzelnen Projekten existieren.

Ein großer Nachteil vieler oben vorgestellter Programme ist, dass sie auf Bibliotheken von Drittanbietern zurückgreifen oder diese erweitern. Gerade dieser Punkt hat den Einsatz dieser Projekte erschwert. Bei einigen gab es Versionsprobleme, andere wurden gar nicht mehr angeboten und wieder andere sind nicht frei erhältlich. Zu letzteren gehört LEDA, welches in nahezu jedem Projekt eingesetzt wurde. Die meisten Programme waren zudem ausschließlich für Linux

erhältlich. Das setzt zum einen ein solches Betriebssystem voraus, zum anderen sind hier oft unzählige Konfigurationen und Installationen notwendig, um das Programm überhaupt zum Laufen zu bringen. Alle diese Stolpersteine haben dazu geführt, dass viele der vorgestellten Projekte keinem Praxistest unterzogen werden konnten. Keine Regel ohne Ausnahme. Diese wurde in *DAJ* und *ViSiDiA* gefunden. Beide Projekte sind in Java realisiert und sind somit plattformunabhängig. Die Installation und das Ausführen dieser Programme ist auch einfach und schnell durchführbar. Bei beiden genügt es ein Zip-Archiv zu entpacken. Das Starten von *DAJ* geschieht über einen Doppelklick auf die gelieferte Jar-Datei. Bei *ViSiDiA* genügt das Aufrufen eines Shell-Skripts.

Ein weiterer Vergleichspunkt ist die Art, wie der Anwender verteilte Algorithmen mit den einzelnen Projekten erzeugen kann. Bei einigen kann der Anwender auf eine bekannte Sprache wie C, C++ oder Java zurückgreifen, in der er schon Programmiererfahrung sammeln konnte und für die er seine favorisierte Programmierumgebung verwenden kann. Andere wiederum verwenden eine eigene Beschreibungs- oder Skriptsprache, mit der sich der Benutzer erst einmal vertraut machen muss.

Auch die Art der Visualisierung ist ein guter Vergleichspunkt. Einige der oben vorgestellten Programme verwenden hierfür fremde Bibliotheken, andere beschränken sich auf eine einfache Textausgabe. Wie bei *ns* erkennbar, muss die Visualisierung auch nicht zeitgleich mit der Simulation geschehen, sondern kann erst nach deren Ablauf angesetzt werden.

Bei der Simulation selbst gibt es kaum Unterschiede. Die meisten sind ereignis- bzw. nachrichtengesteuert oder prozessorientiert. Eine genaue Unterscheidung dieser zwei Typen findet sich bei Sinclair (vgl. [Sinclair02] Seite 12ff) :

1. Ereignisgesteuerte Simulation: ein Ereignis ist eine Zustandsänderung des Modells. Diese Änderung benötigt keine Zeit. Das Modell entwickelt sich aus einer Folge solcher Ereignisse. Um die Entwicklung des Modells zu beschreiben, muss bekannt sein, wann ein Ereignis auftritt und welchen Einfluss es auf das Modell nimmt. Damit dies der Fall ist, werden alle Ereignisse in einer (Ereignis-Zeit) Menge aufgelistet. Die daraus entstandene Liste bildet das Herz einer ereignisgesteuerten Simulation. Zusätzlich werden verschiedene Ereignistypen definiert, um unterschiedliche Ereignisse auszulösen.

2. Die prozessorientierte Simulation: sie umschließt die ereignisgesteuerte mit einer Reihe von Merkmalen, die ein deutlich höheres Niveau an den Tag legen und die das Schreiben, Debuggen und Verstehen eines Simulationsmodells verständlicher machen. Ereignisse, die in irgendeinem Verhältnis zu einander stehen, werden zu einem Prozess zusammengefasst. Die Ausführungszeit eines solchen Prozesses dauert in der Regel mindestens einen

Zeitschritt. Zusätzlich verfügt der Prozess über einen Zustand, der seine momentane Ablaufphase beschreibt. Im Gegensatz zur ereignisgesteuerten Simulation ist die Steuerung der prozessorientierten Simulation wesentlich komplizierter, da hier Funktionen zum Erschaffen, Auslaufen, Beenden oder Pausieren von Prozessen gegeben sein müssen. Hinzu kommen Funktionen, die benötigt werden, um eine Kommunikation oder Synchronisation zwischen mehreren Prozessen zu ermöglichen. Dieser prozessorientierte Ansatz erinnert stark an die Funktionsweise von Betriebssystemen, die ebenfalls mehrere Prozesse verwalten müssen.

DisASTer lässt sich am besten mit den beiden Java-Lösungen *DAJ* und *ViSiDiA* vergleichen. Wobei ein entscheidender Unterschied auftritt. Im Gegensatz zu DisASTer werden die Algorithmen in den beiden anderen Programmen in einer einzigen Klasse definiert, in der das Verhalten der einzelnen Prozesse beschrieben wird. Bei DisASTer entscheiden die Prozesse selbst, wie sie auf welche Ereignisse reagieren und werden dadurch nicht von aussen gesteuert. Dieser Punkt spiegelt die Innovation von DisASTer wieder, denn auf diese Weise können verschiedene Algorithmen, die beispielsweise in Pseudocode vorliegen, 1:1 in Java portiert werden, da sich dieser Code in der Regel auf das Verhalten der einzelnen Prozesse und nicht auf das Verhalten des ganzen Systems bezieht. Konfigurationsmöglichkeiten wie der Topologieeditor in DisASTer finden sich in ähnlicher Form bei *DAP* und *ViSiDiA*. Mit Ausnahme von *ns* besitzen nahezu alle anderen Projekte eine ähnliche Form der Simulation, mit wenigen Abweichungen, wie z.B. die Verwendung mehrerer Threads.

10 Zusammenfassung und Ausblick

Es heißt ja: „Nomen est Omen“. Dieser Ausspruch trifft allerdings nicht auf diese Diplomarbeit zu. Die Probleme bei der Entwicklung von DisASTer hatten zu keinem Zeitpunkt katastrophale Ausmaße angenommen. Auch das Programm selbst wird diesem Namen nicht gerecht, denn es läuft stabil und zuverlässig.

Die Arbeit an diesem Projekt war sehr interessant, da sie viele verschiedene Bereiche abgedeckt hat. Ein erster wesentlicher Aspekt war der Entwurf, in dem einige Entwurfmuster eingesetzt wurden, die viele Problemstellungen der objektorientierten Programmierung vereinfachen konnten. Dadurch konnten die aus vorherigen Vorlesungen erlangten Kenntnisse über objekorientierte Analyse und objektorientiertes Design mit Hilfe von UML noch einmal aufgefrischt und vertieft werden.

Besonders facettenreich war die Arbeit mit Java. In dem entstandenen Programm konnten viele nützliche Techniken eingebracht werden, auch solche, die bisher kaum oder gar nicht bekannt waren. Unter diese Techniken fallen:

1. *Reflection*-API : die es ermöglicht, Klassen aus Datein einzulesen und daraus Instanzen zu bilden oder bestimmte Methoden eines Objekts aufzurufen, wodurch ein Pseudo-Polymorphismus realisiert werden konnte.

2. Swing, grafische Oberflächen und die *2D Graphics*-API : Ein wesentlicher Bestandteil der Arbeit war es, eine ansprechende Oberfläche zu schaffen. Dies ist auch unter Zuhilfenahme einiger einfacher Regeln gelungen. Zusätzlich wurde mit Look-and-Feels gearbeitet, wodurch eine Auffrischung der Erfahrungen aus dem Praxissemester erfolgte. Besonders hervorzuheben ist der Einsatz der *2D Graphics*-API, die viele nützliche Elemente bereitstellt, um die Visualisierungen des Programms zu realisieren.

3. Multithreading : Die Verwendung von Animationen führte zum Einsatz mehrerer Threads, wodurch das Thema Multithreading und Threadsicherheit wieder aufgegriffen, wiederholt und vertieft werden konnte.

Neben den Erkenntnissen in objektorientierter Programmierung und speziell in Java hatte die Arbeit aber auch einen gewissen theoretischen Anspruch durch die intensive Untersuchung der verteilten Algorithmen. Zusätzlich zur genauen Analyse der Algorithmen wurden viele für das entstandene System implementiert.

Abschließend soll noch ein kleiner Ausblick auf die Zukunft von DisASTer geworfen werden. Die eigentliche Entwicklung ist erfolgreich abgeschlossen worden. Es werden aber mit Sicherheit im Laufe der Zeit noch einige Optimierungen und Veränderungen vorgenommen, die entweder vom Entwickler selbst oder im Rahmen von Projektarbeiten durchgeführt werden. Zu den möglichen Änderungen

gehören unter anderem die in der Anforderungsanalyse spezifizierten Nice-to-Haves:

1. Exportieren der Visualisierungen als Film, z.B. in Form von .avi-Dateien. Diese Aufgabe wurde bereits untersucht und im *Java Media Framework*[16] wurde evtl. schon eine Lösung gefunden.

2. Erweitern der Visualisierung um neue Renderer.

3. Komplettierung des Index der Online-Hilfe.

4. Realisierung neuer Algorithmen oder Verbesserung und Erweiterung der bestehenden Beispiele.

Vor allem aber wird DisASTer einen Platz in der Vorlesung „verteilte Systeme“ bekommen und dort Studenten helfen, das Thema „verteilte Algorithmen“ besser zu üben und zu verstehen.

[16] http://java.sun.com/products/java-media/jmf

Literaturverzeichnis

[Coulouris et. al. 02] G. Coulouris, J. Dollimore, T. Kindberg (2002): *Verteilte Systeme - Konzepte und Design*, Addison-Wesley, ISBN 3-8273-7088-1

[Gamma et. al. 96] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1996): *Entwurfsmuster*, Addison-Wesley, ISBN 3-89319-950-0

[Lentzsch03] K. Lentzsch (2003): *Erste Hilfe für Swing - die göbsten Schnitzer in Swing-Oberflächen vermeiden*, Software & Support Verlag, Javamagazin 11/03

[Oechsle02] R. Oechsle (2002): *Verteilte Algorithmen*, Kapitel 9 aus dem Skript „Verteilte Systeme“ an der Fachhochschule Trier, Fachbereich Angewandte Informatik

[Bauderon et. al. 03] M. Bauderon, M. Moshbah, Y. Métevier, A. Zemmari, A. Sellami, B. Derbel, R. Ossamy (2003): *The ViSiDiA Project: Visualization and Simulation of Distributed Algorithms*, http://www.labri.fr/Recherche/LLA/visidia

[Jump93] J. R. Jump (1993): *YACSIM - Reference Manual*, http://www.owlnet.rice.edu/ẽlec428/yacsim/yacsim.man.ps

[Kaynar03] Dilsun Kirli Kaynar (2003): *IOA Language and Toolset*, http://theory.lcs.mit.edu/tds/ioa/

[Lydian] Lydian: *An Extensible Educational AnimationEnvironment for Learning Distributed Algorithms*, http://www.cs.chalmers.se/l̃ydian/

[McCanne, Floyd 03] S. McCanne, S. Floyd (2003): *The Network Simulator - ns-2*, http://www.isi.edu/nsnam/ns/

[Schreiner98] W. Schreiner (1998): *DAJ – A Toolkit for the Simulation of Distributed Algorithms in Java*, http://www.risc.uni-linz.ac.at/software/daj/report/

[Sinclair02] J.B. Sinclair (2002): *Simulation of Computer Systems and Computer Networks: A Process-Oriented Approach*, http://www.owlnet.rice.edu/ẽlec428/yacsim/book.ps

[Sun03] Sun (2003): *Javahelp, A New Standard for Application Help and Online Documentation*, http://java.sun.com/products/javahelp

[Terrevoli97] P. Terrevoli (1997): *Distributed Systems Platform (DSP)*, http://www.dis.uniroma1.it/ alcom-it/proprog/node12.html

[Weiss et. al. 99] B. Weiss, G. Gridling, U. Schmid, K. Schossmaier (1993): *The SimUTC Fault-Tolerant Distributed Systems Simulation Toolkit*, http://citeseer.nj.nec.com/.../weiss99simutc.pdf

[Zaroliagis02] C. Zaroliagis, G. Prasinos, A. Poulakidas, I. Chatzigiannakis (2002): *Distributed Algorithms Platform: Basic Design, Implementation and Functionality*, http://www.ru1.cti.gr/LEP-DAP/papers/basic-design-and-impl.ps.gz

A Klassenübersicht

disaster

- DisASTer.class

disaster.control

- Controller.class
- ControllerState.class
- ControllerStateListener.class
- Manual.class
- Paused.class
- Running.class
- Stopped.class

disaster.examples

- BullyAgent.class
- ByzantinicGeneral.class
- ByzantinicTraitor.class
- CentralServerExclusionAgent.class
- LamportClockAgent.class
- MaekawaAgent.class
- MutualExclusionAgent.class
- RingExclusionAgent.class
- RingLeader01Agent.class
- RingLeader02Agent.class
- RIPAgent.class
  - RoutingTable.class

- SnapshotAgent.class
- TotalPrimAgent.class
- TotalTimeControlledAgent.class
- UniqueClockAgent.class
- VectorClockAgent.class

disaster.gui

- AgentConsoleInfoFrame.class
- AgentMessageQueueInfoFrameGroup.class
  - OpenAllInfoFramesAction.class
- AgentPane.class
  - AddAgentAction.class
  - AddInitMessageAction.class
  - CopyAgentAction.class
  - MessageReceiverCellRenderer.class
  - MoveAgentDownAction.class
  - MoveAgentUpAction.class
  - PasteAgentAction.class
  - ReceptionTimePane.class
  - RemoveAgentsAction.class
  - RemoveInitMessageAction.class
- BackButton.class
  - BackButtonAction.class
  - ListMouseHandler.class
  - ListMouseMotionHandler.class
- ConsoleInfoPane.class
- ControlPane.class
- DefaultInfoFrameAction.class

- Desktop.class
- InfoFrame.class
- InfoFrameGroup.class
  - OpenAllInfoFramesAction.class
- MainFrame.class
- Menubar.class
- MessageQueueInfoFrame.class
  - AgentComponentAction.class
  - AgentMessageQueueCellRenderer.class
  - MessageQueueInfoFrameAction.class
- PreferencesPane.class
- SentMessagesInfoFrame.class
  - SentMessagesInfoFrameAction.class
- Splashscreen.class
  - Closer.class
  - SplashDialog.class
  - SplashWindow.class
- StatusBar.class
- ToolBar.class
- TopologyEditor.class
  - ConnectChosenTool.class
  - ConnectTool.class
  - DisconnectTool.class
  - MoveTool.class
  - Tool.class
- TopologyPane.class

  - AddTopologyManagerAction.class
  - ConnectChosenToolAction.class
  - ConnectToolAction.class
  - DisconnectToolAction.class
  - DoPositionAction.class
  - MoveToolAction.class
  - RemoveAllConnectionsAction.class
- ViewInfoFrame.class
  - ScreenshotAction.class
- ViewPreferences.class
  - AddRendererAction.class
  - AddViewAction.class
  - MoveDownAction.class
  - MoveUpAction.class
  - RemoveRendererAction.class
  - RemoveViewAction.class

disaster.gui.actions

- DeleteMessageAction.class
- EditMenuAction.class
- ExitAction.class
- FileMenuAction.class
- HelpMenuAction.class
- LaFMenuAction.class
- MoveDownAction.class
- MoveUpAction.class
- NewAction.class
- NextAction.class

- NextStepAction.class
- OpenAction.class
- OptionsMenuAction.class
- PauseAction.class
- PlayAction.class
- PreviousAction.class
- SaveAction.class
- SaveAsAction.class
- ShowAboutDialogAction.class
- ShowHelpAction.class
- ShowPreferencesAction.class
- ShowViewPreferencesAction.class
- StopAction.class
- ViewMenuAction.class

disaster.gui.util

- ClassFileChooser.class
- ClassFileFilter.class
- DSEFileFilter.class
- GIFFileFilter.class
- JPEGFileFilter.class
- MessageListTableModel.class
- PNGFileFilter.class

disaster.model

- Agent.class
- AgentComponent.class
- AgentLog.class
- AgentLogEntry.class
- AgentLogListener.class
- CompleteTopology.class
- Connection.class
- EnvironmentChangeListener.class
- Environment.class
- EnvironmentState.class
- EnvironmentUpdateListener.class
- History.class
- HistoryEntry.class
- InitMessage.class
- Message.class
- MessageQueueChangeListener.class
- MessageQueue.class
- Request.class
- Response.class
- StateCopyFactory.class
- TimeoutMessage.class
- TokenRing.class
- TokenRingConnection.class
- Topology.class
- TopologyManager.class

disaster.view

- AgentRenderer.class
- DefaultAgentRenderer.class
  - AgentPreferencesPane.class
- Renderer.class
- SequenceViewAgentRenderer.class
  - AgentRendererPane.class
    - ButtonCellEditor.class
    - ChangeRendererAction.class
    - LevelUpAction.class
- SequenceView.class
- SequenceViewClockRenderer.class
- SequenceViewConstants.class
- SequenceViewMessageRenderer.class
  - MessagePreferencesPane.class
- TopologyViewAgentRenderer.class
  - AgentRendererPane.class
    - ButtonCellEditor.class
    - ChangeRendererAction.class
    - LevelUpAction.class
- TopologyView.class
- TopologyViewConnectionRenderer.class
  - ConnectionPreferencesPane.class
  - StrokeCellRenderer.class
- TopologyViewMessageRenderer.class
  - MessagePreferencesPane.class
- View.class
- ViewUtils.class